SERIALIZABILITY AND HETEROGENEOUS TRUST FROM TWO PHASE COMMIT TO BLOCKCHAINS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Isaac Cameron Sheff

August 2019

SERIALIZABILITY AND HETEROGENEOUS TRUST FROM TWO PHASE COMMIT TO BLOCKCHAINS

Isaac Cameron Sheff, Ph.D.

Cornell University 2019

As distributed systems become more federated and cross-domain, we are forced to rethink some of our core abstractions. We need heterogeneous systems with rigorous consistency and self-authentication guarantees, despite a complex landscape of security and failure tolerance assumptions. I have designed, built, and evaluated heterogeneous distributed algorithms with broad applications from medical privacy to blockchains.

This dissertation examines three novel building blocks for this vision.

First, I show that serializable transactions cannot always be securely scheduled when data has different levels of confidentiality. I have identified a useful subset of transactions that can always be securely scheduled, and built a system to check and execute them.

Second, I present Charlotte, a heterogeneous system that supports composable Authenticated Distributed Data Structures (like Git, PKIs, or Bitcoin). I show that Charlotte produces significant performance improvements compared to a single, universally trusted blockchain.

Finally, I develop a rigorous generalization of the consensus problem, and present the first distributed consensus which tolerates heterogeneous failures, heterogeneous participants, and heterogeneous observers. With this consensus, cross-domain systems can maintain ADDSs, or schedule transactions, without the expensive overhead that comes from tolerating the sum of everyone's fears.

BIOGRAPHICAL SKETCH

Isaac Sheff was born on Christmas Eve 1989 to Kimberly and David Sheff, in Mercy Hospital of Iowa City. In 1993, his family moved to Hamden, Connecticut, where his younger siblings Benjamin and Olivia were born. Isaac attended kindergarten through second grade at Bear Path Elementary. Redistricting moved him to Alice Peck Elementary before third grade, and he switched to Hamden Hall Country Day School in fourth grade.

In 2003, Isaac and his family left their friends in Connecticut behind, and returned to Iowa City. Isaac attended eighth grade at Northwest Junior High, and graduated in 2008 from West High School. In 2007, Isaac attended Cy-Tag, a three week intensive at Iowa State University, where he first learned to program. At West High, Isaac discovered a passion for engineering and mathematics, joining FIRST robotics team 167, and math-themed clubs including Fun For Friday (F^3), and the $\mu\alpha\theta$ and American Regions Math League teams.

In 2008, Isaac moved to Pasadena, California, and enrolled in the California Institute of Technology, earning a Bachelor of Science (majoring in Computer Science) in 2012. Isaac arrived in California intending to build the next generation of spacecraft, but quickly became sidetracked by the plethora of sciences available. By junior year, it had become apparent that Isaac's programming skills and passion for engineering and mathematical problem solving served him well in Computer Science.

Isaac spent the summer of 2012 as a Software Engineering Intern at Google Los Angeles. While the challenges presented each day at Google were interesting, Isaac feared he would soon grow bored without the opportunity to work on new, unsolved problems. In the fall of 2012, Isaac enrolled in Cornell's Doctor of Philosophy program in Computer Science to do just that.

For my parents

For my partner

ACKNOWLEDGMENTS

Some of my loneliest days have been as a Ph.D. student, but science is not a solitary activity. It is only with the collaboration and assistance of others that I have found any success.

First, I must thank my advisor, Andrew Myers, for his continuous support and collaboration, his supply of ideas and context, and his guidance through this process. Andrew has taught me so much about how to research, from formulating ideas all the way through to communicating results through papers and presentations. In his research group, *Applied Programming Languages,* I have found collaborations without whom I could not have completed this work.

Next, I must thank my "other advisor" of sorts. Robbert van Renesse has been indispensable for his insight and understanding in Distributed Systems, both the subject matter and the community, as well as his professional advice. He was the first person with the good sense to drag me to meet people at a conference, and the first person to explain consensus to me in a way I understood. I could never have finished this without him.

Cornell's out-of-field minor requirement is unusual, and when I set out to try something truly different, Oren Falk had the grace, patience, and good humor to take me up on it. My out-of-field minor is in Medieval Studies, not because it's related to this dissertation, but because I personally find it enriching. It may also have improved my reading speed and comprehension, and general writing skills. Despite my lack of qualification for graduate-level historical studies, Oren guided me through some fascinating courses in Medieval Violence and Anglo-Saxon England, and encouraged me to explore Cornell's other offerings in the subject. I am truly grateful for what has been an exciting journey, and a welcome complement to my research. Thank you, Oren.

I want to thank Mark Moir and Maurice Herlihy. At Oracle, Mark, Maurice, and I stumbled into the bizarre world of blockchains together, and it was in our discussions that the first conceptions of Charlotte came to be. Without them, this dissertation would be almost, if not entirely, different.

I especially want to thank my co-authors, including Andrew Myers and Robbert van Renesse, but also Tom Magrino, Xinwen Wang, Jed Liu, Haobin Ni, and Zhenjia Xu. None of these projects could have been completed without them, and not only because most of the implementation work on Secure Scheduling, Heterogeneous Consensus, and Charlotte Nakamoto is theirs.

I want to thank the colleagues with whom I've taught courses, including Fred Schneider, Deniz Altinbuken, Lucja Kot, David Gries, and Ken Birman. Teaching is an integral part of a Ph.D., and I've learned it entirely through experience by your side.

Research is a collaborative effort, and not only with co-authors. Thank you to everyone who has helped me with research ideas, editing, practice talks, and late-night deadlines, including but not limited to: Josh Acay, Soumya Basu, Ken Birman, Ethan Cecchetti, Natacha Crooks, Andrew Hirsch, Matthew Milano, Laure Thompson, Fabian Muehlboeck, Rolph Recto, and Drew Zagieboylo.

I could never have completed a Ph.D. without help not only professionally, but personally. I'd like to thank my friends throughout this chapter of my life, including but not limited to: Gabriel Bender, Eleanor Birrell, Marin Cherry, Tawny Cuykendall, Alex Fix, Elisavet Kozyri, Stavros Nikolaou, Brittany Nkounkou, Karthik Raman, Xanda Schofield, Daniel Schroeder, Eston Schweickart, Karn Seth, Ruben Sipos, Adith Swaminathan, Sidharth Telang, and Scott Wehrwein.

Finally, I'd like to thank Becky Stewart, Jessica Beeve, Tammy Gardner, Lacy Lucas, Corey Torres, and the rest of the Cornell CS staff, without whom my Ph.D. experience would have been literally impossible.

I have received funding through generous gifts from from Oracle and Ripple, as well as grants from the NSF.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Distributed Systems have become an integral part of not only modern computing, but modern life. Every day, billions of people each execute dozens of transactions involving massive, geodistributed datastores including those run by Google [39], Facebook [24], and Amazon [45]. These systems promise to keep data available, consistent, and secure.

However, they don't always succeed. Distributed systems crash when individual components fail, and the rest of the system fails to compensate. They are inconsistent when different users see different values, or different orders of events. They leak data when it is sent to unauthorized parties, or taint it when unauthorized parties insert malicious values.

1.1 Ideal Distributed Systems

Abstractly, distributed systems deal with some set of *participants* (sometimes called "nodes," "processors," or "processes"), usually computers, which execute computation and send *messages* to each other over the *network*. *Observers* are abstract entities that observe messages in the system and make demands about system properties. Observers define the requirements on the system: how many failures it needs to tolerate, and what tasks it must perform under what circumstances. Participants are merely components of the system, although some entities (such as people) can be both observers and participants. Not all systems always satisfy all observers. There are some traditional assumptions about this setting. For instance, we usually assume that once a message is sent, it will eventually

arrive [81]. Often we assume there is no way to tell how long a message will take, and so detecting when a participant has failed is difficult [33].

I believe a Distributed System should be like a well-run kitchen. Cooks and tools are participants, and they pass messages in the form of ingredients in various stages of preparation. Diners are observers: they demand that tasks be done, and have opinions about the results in terms of availability and food safety. I believe in three core principles that apply to both kitchen organization and distributed system design: least ordering, fault tolerance, and heterogeneity.

1.1.1 Least Ordering

Participants should not be forced to do tasks in any particular order, unless something inherent to those tasks requires it. This is an old concept in the database community [19], and a good rule of thumb for a kitchen as well.

For example, suppose a naive cook has a recipe for a meal involving the steps:

- Bake the bread
- Simmer the soup
- Chop the salad

A linear approach would be to put the bread in the oven until it's done, and afterwards simmer the soup until it's done, and afterward chop the salad. However, such an approach takes unnecessarily long, and doesn't allow for other scheduling constraints (such as trying to finish multiple foods at mealtime).

On the other hand, an experienced cook can put the bread in the oven, and while it's baking, set the soup to simmer, and while it's simmering, chop the salad. These tasks are not inherently ordered, and so a well-run kitchen can do them however is most efficient.

As another example, imagine a distributed system for a bank. If Alice transfers money to Bob, and Carol transfers money to Dave, there is no reason for the two transfers to be ordered: they can even be handled by totally separate participants. On the other hand, if Eve tries to send all her money to Fred and also to Gloria, it matters which transaction happens first: that controls who gets the money.

1.1.2 Fault Tolerance

Some participants are going to fail, and a distributed system should complete tasks anyway. For example, suppose a naive cook goes to chop a salad, and the knife is dull. In distributed systems terms, this is a crash failure: something stopped working. Without any failure tolerance, the naive cook cannot complete the salad. On the other hand, a well-run kitchen should have multiple knives. An experienced cook can use a back-up knife, and complete the salad.

As another example, consider data storage. If data is stored on one machine, it is lost when that machine fails. However, if it is backed up onto multiple machines, it can remain available so long as one of them still works.

1.1.3 Heterogeneity

Most modern distributed system design is focused on *homogeneous* systems, with respect to participants, failures, and *observers*. However, some of the most successful distributed systems, including the internet itself, are extremely heterogeneous.

Participants

Not all participants have the same privileges and capabilities. A distributed system should respect these differences. Most traditional distributed system or algorithm designs, such as paxos [83, 84], PBFT [30], Chord [141], Bittorrent [37], or even Bitcoin [106] simply assume some collection of undifferentiated participants. Some systems, including Fabric [90] and DStar [155], recognize that different participants have different abilities and restrictions.

For example, there are messages which should flow to some participants, but not others. In a kitchen we can imagine soup pots and ovens as participants, but there are ingredients, such as broth, that should flow into the soup pot, but not into the oven.

As another example, most kitchens are willing to tolerate more knife failures than oven failures. Perhaps because ovens are more expensive than knives, they are willing to stop making certain dishes if a small number of ovens fail, but many knives would have to fail before the menu would change.

In distributed systems, participants are often heterogeneous for security reasons: some machines might not be trusted to know some data, or to influence some data. For example, a hospital might allow some employees' computers to access

patients' addresses, but not their HIV status. In chapter 2, we dig into examples where security in a heterogeneous system can be deceptively difficult to guarantee.

Participants can also be heterogeneous for failure-tolerance reasons. Suppose an application is designed to tolerate the failure of one hosting provider. It should be available whenever one of Google or Amazon or Microsoft fail, and it's using multiple computers in each. For this application, 6 Google machines failing is crucially different than 3 Google machines and 3 Microsoft machines failing. In chapter 4, we develop an expressive framework for detailing nuanced failure tolerance policies.

Failures

Not all failures are the same. Distributed Systems should be as specific as possible in defining their failure tolerances, especially if that means tolerating a mix of different types of failures.

For example, there is a critical difference between an oven that has crashed (stopped working), and one that is byzantine (it behaves arbitrarily). A cook will be unable to put a lasagna in a crashed oven, and so it will never arrive at the customer. A byzantine oven, however, might accept the lasagna, but undercook it, and make the customer ill.

There are a number of failure types in distributed systems literature [120, 86, 59, 5, 87], but most systems only consider one. It is popular, for example, to describe a system as tolerating at most f byzantine failures out of n participants. It doesn't matter which of the n participants fail, and no other failure types are considered. Systems which do consider multiple failure types are said to have a

mixed failure model [134]. In chapter 4, we describe a consensus algorithm that embraces and extends a mixed failure model.

Observers

Not all principals observing a distributed system make the same assumptions. A distributed system should take this into account, and provide guarantees that are specific to each observer. Most traditional distributed systems, including virtually all databases, are built on some universal set of assumptions about failure tolerances, capabilities, and requirements. A few, such as Stellar [96] and Cobalt [94], directly embrace the notion that not everyone will have the same expectations.

For example, we can consider diners the observers of a kitchen. One diner might prefer an overcooked lasagna (in technical terms, a safety failure) to one that never arrives (a liveness failure). Another diner might prefer that the lasagna never arrive.

Another example occurs when multiple parties try to collaborate on a data structure. If two banks, for instance, are trying to agree on a common ledger of financial transfers, bankers at one bank might have a different idea about which computers are trustworthy than bankers at the other bank. In chapter 3, we discuss cross-domain data structures and how to express their integrity properties. In chapter 4, we explain how to achieve consensus among heterogeneous observers.

1.1.4 Building With These Ideals

In my research, I have tried to adapt existing systems, and build new ones in line with these ideals. Existing systems which fail to embrace least ordering, failure tolerance, or heterogeneity often pay a price in performance or security.

1.2 Security

1.2.1 Information Flow Control (IFC)

Distributed systems are perpetually at risk of sending data to unauthorized parties, or allowing unauthorized actors undue influence. This is especially true when participants are heterogeneous, and not all participants can be trusted with all data.

Information Flow Control (IFC) is a set of techniques for ensuring systems do not leak information (confidentiality) [46] or allow unauthorized influence (integrity) [22, 118]. IFC enforcement guaranties *non-interference*: the idea that an unauthorized party should not be able to distinguish two executions of a program, so long as the inputs they're authorized to see are the same.

1.2.2 Failure Tolerance

Security can also include fault-tolerance: avoiding system failures when participants fail. This is especially evident for applications like power grids, where malicious participants causing a blackout are a real-world security concern.

Lantian Zheng demonstrated that the same IFC techniques could apply to availability as well, allowing data labels to specify the conditions under which the data must remain available [158, 157, 159]. In this unified view, each data label specified its own failure tolerance conditions in terms of Confidentiality, Availability, and Integrity, a traditional "triad" for expressing information security. In fact, with Fabric-style labels featuring policy *owners*, it is possible to address the security requirements of each datum *for each observer*. My work on Observer Graphs (§ 4.3) grows directly from this material and attempts to express nuances that Lantian's labels could not. For example, our labels can describe situations in which several groups of observers disagree with each other on the value of a datum, but within each group, observers agree on the value.

1.3 Transactions

In a distributed setting, it is extremely difficult for programmers to account for the possibility of many programs running at the same time. Each *transaction* is an execution of a distributed program, generally thought of as running concurrently with other transactions. A transaction might, for instance, subtract from the balance of one bank account, and add to the balance of another, on another participant.

While a dizzying array of scheduling options are available [143, 107, 125, 17, 146, 92, 79, 41, 6, 14, 102], *serializability* remains the gold standard. Serializable transactions[1] behave as though nothing else is running on the system at the time.

[1] similar to ACID or Strongly Consistent transactions

For any allowed system behavior, there is a serial order of transactions which would produce the same behavior (if they ran fast enough) [107, 125].

Achieving serializability, however, can be deceptively tricky. All participants involved in a transaction have to engage in some kind of scheduling protocol to ensure events are ordered consistently. By far the most popular scheduling protocol is 2 Phase Commit (2PC) [19], but as I show in chapter 2, 2PC can leak information in heterogeneous systems, even if each transaction itself is secure. In fact, it is impossible to securely serialize arbitrary secure transactions. I do, however, identify a class of securely serializable transactions, and develop a protocol, Staged Commit, to securely serialize them. We modified the Fabric [90] compiler and runtime to use Staged Commit.

1.4 Consensus

Unfortunately, neither 2PC nor Staged Commit are as failure-tolerant as we might like. When some participants can crash, the protocol may never complete, and transactions will not be scheduled.

Consensus, in a nutshell, is the problem of getting observers to agree on a value (such as the order of transactions), despite the possibility that some portion of the participants may fail (e.g. crash or behave maliciously). Unfortunately, it is impossible to guarantee that any protocol for achieving consensus will terminate (finishe) without assuming more of the network than simply "any message sent is eventually delivered" [54]. The most famous solution is Paxos [83, 84, 83, 145], which assumes the network is *semi-synchronous*: there is some bound, possibly unknown, on the time between when a message is sent and when it's delivered.

Practical Byzantine Fault Tolerance, the most famous consensus algorithm tolerating byzantine failures, relies on a similar assumption [30]. Some other consensus protocols rely on random number generation to provide a probabilistic termination guarantee instead [103, 3].

Consensus has other known limitations. For example, when participants are homogeneous, there must be greater than $2f$ participants to tolerate f crash failures, and greater than $3f$ to tolerate byzantine failures [86]. In chapter 4, I generalize these limitations to the heterogeneous setting: wherein participants, observers, and failures are all heterogeneous. In § 4.8, I explore scenarios in which protocols that take heterogeneity into account can subvert the limit of homogeneous protocols, avoiding the costs associated with unnecessary participants.

1.5 Blockchains

Recently, *blockchains* have stirred up a great deal of interest in serializing transactions using large-scale consensus with diverse participants. A blockchain is a data structure wherein each datum, or *block*, refers to the previous using a collision-resistant hash, and no two blocks in the chain can refer to the same predecessor. They are traditionally used as an append-only *log* or *ledger*, and require a consensus mechanism to ensure only one element is appended in each entry. Such a ledger can be used to keep track of money, as in a Cryptocurrency [106], or even the state transitions of an arbitrary state machine, such as a "smart contract" [121, 56].

Most chains use a single, homogeneous consensus protocol to sequentially add transactions to an ever-growing ledger [106, 56]. While these blockchains are built to be failure-tolerant, they largely fail to embrace least ordering and heterogeneity.

1.5.1 Least Ordering

Blockchain transactions tend to be serializ*ed*, not merely serializ*able*. The difference is subtle but important.

- *serializable* transactions behave *as though* they were the only transaction running at the time. The resulting execution must be *equivalent to* an execution with all transactions run in some serial order. The system *can optimize* runtime and resource usage by running independent transactions in parallel, while preserving equivalence to a serial order. Many existing systems take advantage of this optimization, including Postgres [111], MySQL [117], and .NET [1].
- *serialized* transactions each *actually are* the only transaction executing at the time. The resulting execution *actually has* all transactions run in some serial order. The system *can't optimize* anything: the schedule of execution is linear, and totally decided by consensus.

As a result, existing blockchain architectures are woefully slow: their speed is equivalent to a few transactions per second, while traditional architectures can run thousands per second [40].

Several current efforts aim to build blockchains without total serialization using sharding [77, 93, 152, 43, 25, 140, 57, 150], but at best these projects ask "what is the most ordering we can keep," instead of "what is the least ordering we need?" To address this fundamental failure to embrace the Least Ordering principle, I developed Charlotte (chapter 3), a framework for Authenticated Distributed Data Structures (such as blockchains), in which blocks are unordered by default, and ordering mechanisms can be added when necessary.

1.5.2 Heterogeneity

Most blockchain projects, even the sharded ones, rely on a single homogeneous consensus mechanism. They hope to gather sufficiently many participants that that mechanism will be sufficiently trustworthy for all applications. This has two substantial drawbacks.

First, despite tolerating many failures, these consensus mechanisms do not satisfy everyone. For instance, banks usually demand that their data be consistent so long as their computers are working, even if all other computers in the world are not. No homogeneous consensus algorithm will satisfy multiple banks simultaneously.

Second, these consensus algorithms are incredibly expensive. This is part of why blockchain architectures are so slow. Bitcoin-style "Proof-of-Work," in particular, uses an enormous amount of energy, more than several moderate size countries [44].

These shortcomings arise because blockchains have failed to embrace heterogeneity. Rather than developing a single integrity mechanism trustworthy enough for all applications, each application can use its own integrity mechanism. Since most applications are specific to some organization (or individual), these mechanisms can be much faster. Even transactions involving multiple applications need only satisfy the integrity requirements of the applications involved, rather than the whole world. Charlotte (chapter 3) allows for these composable applications, and defines integrity and availability properties in a formal and composable way. Heterogeneous Consensus (chapter 4) can serve as an integrity mechanism

for Charlotte applications, allowing them to tailor their fault tolerance to specific observers.

1.6 Roadmap

In this dissertation, I discuss building distributed systems with serializable transactions. To address the shortcomings of existing distributed systems, and embrace least ordering, fault tolerance, and heterogeneity, I present three related projects: Safe Serializable Secure Scheduling, Charlotte, and Heterogeneous Consensus.

1.6.1 Safe Serializable Secure Scheduling

In chapter 2, I discover security problems with existing transaction scheduling techniques (2PC) in a setting with heterogeneous participants. Scheduling protocols are supposed to impose only necessary ordering between transactions, in keeping with the least ordering principle, but unfortunately, when different participants are permitted to know different data, the scheduling protocol itself can leak information.

I show that no protocol can schedule all possible sets of safe transactions securely. I also develop a new subset of transactions, *relaxed monotonic* transactions, which my new protocol, *Staged Commit* can schedule. We implemented and tested our Staged Commit protocol in the Fabric system [90], and present the results in § 2.8. These results were published at CCS 2016 [128].

1.6.2 Authenticated Distributed Data Structures

In chapter 3, I address the shortcomings in modern blockchains while preserving prized properties like self-authenticating data, and provable commits. Charlotte is a framework for Authenticated Distributed Data Structures (ADDSs) (including blockchains) that embraces least ordering, fault tolerance, and heterogeneity.

Charlotte allows data structures to compose, and provides a formal model for reasoning about availability properties of data structures and their compositions. I demonstrate how existing ADDSs can be replicated within the Charlotte framework, including Git, Bitcoin, and Timestamping, with minimal overhead. I also show that, with Least Ordering, the actual transactions in the Bitcoin payment history could be committed about 70 times faster.

1.6.3 Heterogeneous Consensus

In chapter 4, I design Heterogeneous Consensus, the first consensus algorithm that can be tailored for heterogeneous participants, observers, and failures. I demonstrate the resources this can save in a variety of scenarios, when compared to a homogeneous consensus algorithm. Heterogeneous Consensus can be fitted into Charlotte to create blockchains and other data structures with rich, composable integrity properties. As an example, I demonstrate blockchains using Heterogeneous Consensus and Charlotte.

CHAPTER 2

SAFE SERIALIZABLE SECURE SCHEDULING

Transactions and the Trade-off Between Security and Consistency

Synopsis

Modern applications often operate on data in multiple administrative domains. In this federated setting, heterogeneous participants may not fully trust each other. These distributed applications use transactions as a core mechanism for ensuring reliability and consistency with persistent data. However, the coordination mechanisms needed for transactions can both leak confidential information and allow unauthorized influence. To return to our kitchen analogy (§ 1.1.3), we're seeking to prevent the equivalent of broth flowing into the oven, when it should only be allowed to flow to the soup pot.

By implementing a simple attack, we show these side channels can be exploited. However, our focus is on preventing such attacks. We explore secure scheduling of atomic, serializable transactions in a setting with heterogeneous trust. While we prove that no protocol can guarantee security and liveness in all settings, we establish conditions for sets of transactions that can safely complete under secure scheduling. Based on these conditions, we introduce *staged commit*, a secure scheduling protocol for federated transactions. This protocol avoids insecure information channels by dividing transactions into distinct stages. We implement a compiler that statically checks code to ensure it meets our conditions, and a system that schedules these transactions using the staged commit protocol. Experiments on this implementation demonstrate that realistic federated transactions can be scheduled securely, atomically, and efficiently.

This chapter is based on work published at CCS 2016 [128].

2.1 Introduction

Many modern applications are distributed, operating over data from heterogeneous domains. Distributed protocols are used by applications to coordinate across physically separate locations, especially to maintain data consistency. However, distributed protocols can leak confidential information unless carefully designed otherwise.

Distributed applications are often structured in terms of *transactions*, which are atomic groups of operations (§ 1.3). For example, when ordering a book online, one or more transactions occur to ensure that the same book is not sold twice, and to ensure that the sale of a book and payment transfer happen atomically. Transactions are ubiquitous in modern distributed systems. Implementations include Google's Spanner [39], Postgres [111], and Microsoft's Azure Storage [27]. Common middleware such as Enterprise Java Beans [101] and Microsoft .NET [1] also support transactions.

Many such transactions are distributed, involving multiple heterogeneous participants (vendors, banks, etc.) (§ 1.1.3). Crucially, these participants may not be equally trusted with all data. Standards such as X/Open XA [2] aim specifically to facilitate transactions that span multiple systems, but none address information leaks inherent to transaction scheduling.

Distributed transaction implementations are often based on the two-phase commit protocol (2PC) [52]. We show that 2PC can create unintentional channels

through which private information may be leaked, and trusted information may be manipulated. We expect our results apply to other protocols as well.

There is a fundamental tension between providing strong consistency guarantees in an application and respecting the security requirements of the application's trust domains. This work deepens the understanding of this trade-off and demonstrates that providing both strong consistency and security guarantees, while not always possible, is not a lost cause.

Concretely, we make the following contributions in this chapter:

- We describe *abort channels*, a new kind of side channel through which confidential information can be leaked in transactional systems (§ 2.2).
- We demonstrate exploitation of abort channels on a distributed system (§ 2.2.3).
- We define an abstract model of distributed systems, transactions, and information flow security (§2.3), and introduce *relaxed observational determinism*, a noninterference-based security model for distributed systems (§ 2.3.7).
- We establish that within this model, it is not possible for any protocol to securely serialize all sets of transactions, even if the transactions are individually secure (§ 2.4).
- We introduce and prove a sufficient condition for ensuring serializable transactions can be securely scheduled (§ 2.5).
- We define the *staged commit* protocol, a novel secure scheduling protocol for transactions meeting this condition (§ 2.6).

- We implement our novel protocol in the Fabric system [91], and extend the Fabric language and compiler to statically ensure transactions will be securely scheduled (§ 2.7).
- We evaluate the expressiveness of the new static checking discipline and the runtime overhead of the staged commit protocol (§ 2.8).

We discuss related work further in § 2.9, and conclude in § 2.10.

2.2 Abort Channels

Two transactions working with the same data can *conflict* if at least one of them is writing to the data. Typically, this means that one (or both) of the transactions has failed and must be *aborted*. In many transaction protocols, including 2PC, a participant[1] involved in both transactions can abort a failed transaction by sending an *abort message* to all other participants in the failed transaction [52]. These abort messages can create unintended *abort channels*, through which private information can be leaked, and trusted information can be manipulated.

An abort message can convey secret information if a participant aborts a transaction otherwise likely to be scheduled, because another participant in the same transaction might deduce something about the aborting participant. For example, that other participant might deduce that the abort is likely caused by the presence of another—possibly secret—conflicting transaction.

Conspirators might deliberately use abort channels to covertly transfer information within a system otherwise believed to be secure. Although abort channels

[1]Transaction participants are often processes or network nodes.

communicate at most one bit per (attempted) transaction, they could be used as a high-bandwidth covert channel for exfiltration of sensitive information. Current transactional systems can schedule over 100 million transactions per second, even at modest system sizes [50]. It is difficult to know if abort channels are already being exploited in real systems, but large-scale, multi-user transactional systems such as Spanner [39] or Azure Storage [27] are in principle vulnerable.

Abort messages also affect the integrity of transaction scheduling. An abort typically causes a transaction not to be scheduled. Even if the system simply retries the transaction until it is scheduled, this still permits a participant to control the ordering of transactions, even if it has no authority to affect them. For example, a participant might gain some advantage by ensuring that its own transactions always happen after a competitor's.

Transactions can also create channels that leak information based on timing or termination [11, 20]. We treat timing and termination channels as outside the scope of this work, to be handled by mechanisms such as timing channel mitigation [78, 10, 15]. Abort channels differ from these previously identified channels in that information leaks via the existence of explicit messages, with no reliance on timing other than their ordering. Timing mitigation does not control abort channels.

2.2.1 Rainforest Example

A simple example illustrates how transaction aborts create a channel that can leak information. Consider a web-store application for the fictional on-line retailer Rainforest, illustrated in Fig. 2.1. Rainforest's business operates on data from suppliers, customers, and banks. Rainforest wants to ensure that it takes money from

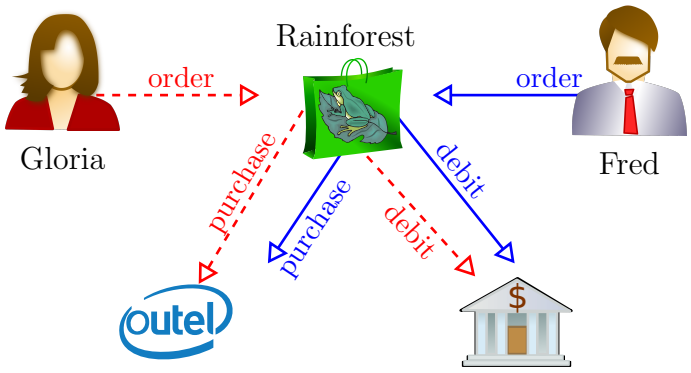

Figure 2.1: Rainforest example. Gloria and Fred each buy an Outel chip via Rainforest's store. Gloria's transaction is in red, dashed arrows; Fred's is in blue, solid arrows.

---

customers only if the items ordered have been shipped from the suppliers. As a result, Rainforest implements purchasing using serializable transactions. Customers expect that their activities do not influence each other, and that their financial information is not leaked to suppliers. These expectations might be backed by law.

In Fig. 2.1, Gloria and Fred are both making purchases on Rainforest at roughly the same time. They each purchase an Outel chip, and pay using their accounts at CountriBank. If Rainforest uses 2PC to perform both of these transactions, it is possible for Gloria's computer to receive an abort when Outel tries to schedule her transaction and Fred's. The abort leaks information about Fred's purchase at Outel to Gloria. Alternatively, if Gloria is simultaneously using her bank account in an unrelated purchase, scheduling conflicts at the bank might leak to Outel, which could thereby learn of Gloria's unrelated purchase.

These concerns are about confidentiality, but transactions may also create integrity concerns. The bank might choose to abort transactions to affect the order

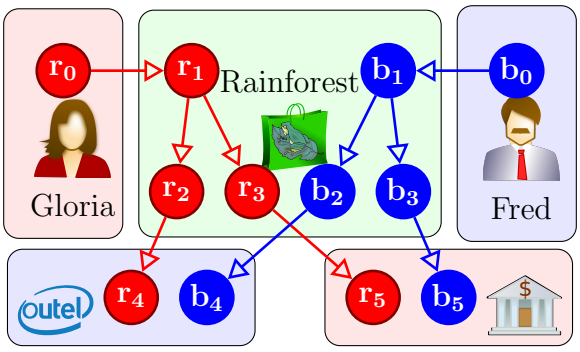

Figure 2.2: The events of the transactions in Fig. 2.1. Gloria's transaction consists of r_0, r_1, r_2, r_3, r_4, and r_5. Bob's consists of b_0, b_1, b_2, b_3, b_4, and b_5. Happens-before ($\rightarrowtriangle$) relationships are arrows. The shaded blocks around events indicate locations, and are labeled with participants from Fig. 2.1.

---

in which Outel sells chips. Rainforest and Outel may not want the bank to have this power.

2.2.2 Hospital Example

As a second, running example, we use two small programs with an abort channel. Suppose Patsy is a trusted hospital employee, running the code in Fig. 2.3a to collect the addresses of HIV-positive patients in order to send treatment reminders. Patsy runs her transaction on her own computer, which she fully controls, but it interacts with a trusted hospital database on another machine. Patsy starts a transaction for each patient `p`, where transaction blocks are indicated by the keyword `atomic`. If `p` does not have HIV, the transaction finishes immediately. Fig. 2.3c shows the resulting transaction in solid blue. (Events in the transaction are represented as ovals; arrows represent dependencies between transaction events.) Otherwise, if the patient has HIV, Patsy's transaction reads the patient's address and prints it (the blue transaction in Fig. 2.3c, including dashed events).

Suppose Mallory is another employee at the same hospital, but is not trusted to know each patient's HIV status. Mallory is, however, trusted with patient addresses. Like Patsy, Mallory's code runs on her own computer, which she fully controls, but interacts with the trusted hospital database on another machine. She runs the code in Fig. 2.3b to update each patient's address in a separate transaction, resulting in the red transaction in Fig. 2.3c. When Mallory updates the address of an HIV-positive patient, her transaction might conflict with one of Patsy's, and Mallory would observe an abort. Thus Mallory can learn which patients are HIV-positive by updating each patient's address while Patsy is checking the patients' HIV statuses. Each time one of Mallory's transactions aborts, private information leaks: that patient has HIV.

One solution to this problem is to change Patsy's transaction: instead of reading the address only if the patient is HIV positive, Patsy reads every patient's address. This illustrates a core goal of our work: identifying which programs can be scheduled securely. In Fig. 2.4a, lines 3 and 4 of Patsy's code have been switched. As Fig. 2.4c shows, both possible transactions read the patient's address. Since Mallory cannot distinguish which of Patsy's transactions has run, she cannot learn which patients have HIV.

2.2.3 Attack Demonstration

Using code resembling Fig. 2.3, we implemented the attack described in our hospital example (§ 2.2.2) using the Fabric distributed system [9, 91]. We ran nodes representing Patsy and Mallory, and a storage node for the patient records.

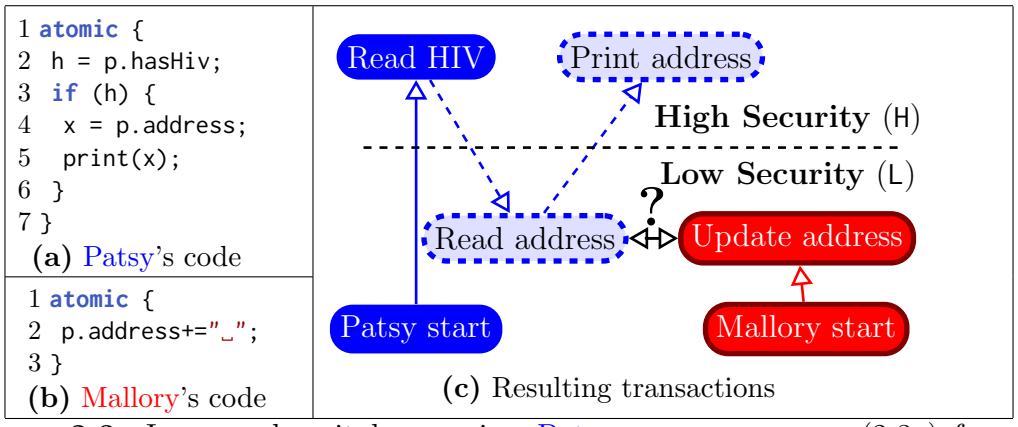

Figure 2.3: Insecure hospital scenario. Patsy runs a program (2.3a) for each patient p. If p has HIV (which is private information), she prints out p's address for her records. The resulting transaction takes one of two forms. Both begin with the event Patsy start. If p is HIV negative, the transaction ends with Read HIV. Otherwise, it includes the blue events with dashed outlines. Meanwhile, Mallory updates the p's (less secret) address (2.3b), resulting in the transaction with red, solid-bordered events. This conflicts with Patsy's transaction, requiring the system to order the update and the read, exactly when p has HIV ("?" in 2.3c).

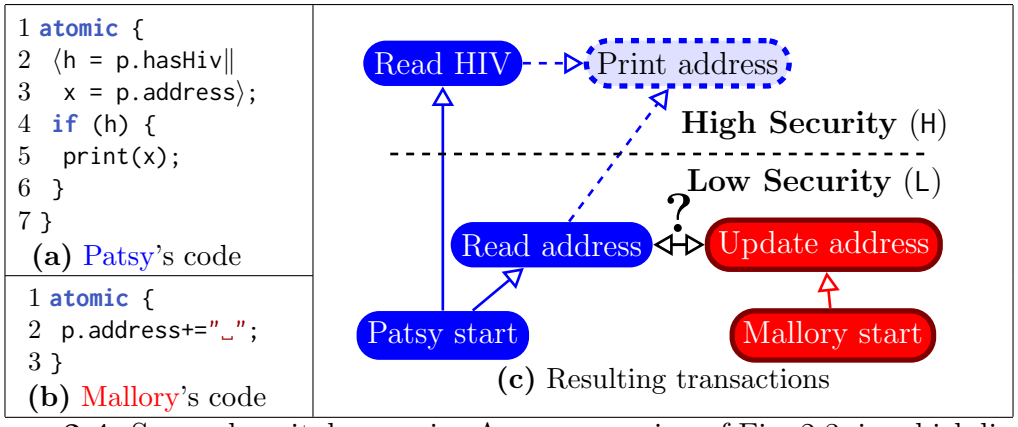

Figure 2.4: Secure hospital scenario. A secure version of Fig. 2.3, in which lines 3 and 4 of Patsy's code (2.3a) are switched, and the resulting lines 2 and 3 can be run in parallel (⟨ ‖ ⟩). Thus the transaction reads p's address regardless of whether p has HIV, and so Mallory cannot distinguish which form Patsy's transaction takes.

To improve the likelihood of Mallory conflicting with Patsy (and thereby receiving an abort), we had Patsy loop roughly once a second, continually reading the address of a single patient after verifying their HIV-positive status. Meanwhile, Mallory attempted to update the patient's address with approximately the same frequency as Patsy's transaction.

Like many other distributed transaction systems, Fabric uses two-phase commit. Mallory's window of opportunity for receiving an abort exists between the two phases of Patsy's commit, which ordinarily involves a network round trip. However, both nodes were run on a single computer. To model a cloud-based server, we simulated a 100 ms network delay between Patsy and the storage node.

Getting this to work was challenging, because Fabric caches its objects optimistically. When Mallory updates the patient's address, it would invalidate Patsy's cached copy, causing Patsy's next transaction to abort and retry. Furthermore, Fabric implements an exponential back-off algorithm for retrying aborted transactions. As a result, we had to carefully tune the transaction frequencies to prevent Mallory from starving out Patsy.

We ran this experiment for 90 minutes. During this time, Mallory received an abort roughly once for every 20 transactions Patsy attempted. As a result, approximately every 20 seconds, Mallory learned that a patient had HIV. In principle, many such attacks could be run in parallel, so this should be seen as a minimal, rather than a maximal, rate of information leakage for this setup.

As described later, our modified Fabric compiler (§2.7) correctly rejects Patsy's code. We amended Patsy's code to reflect Fig. 2.4, and our implementation of the staged commit protocol (§ 2.6) was able to schedule the transactions without leaking information. Mallory was no more or less likely to receive aborts regardless of whether the patient had HIV.

2.3 System Model

We introduce a formal, abstract system model that serves as our framework for developing protocols and proving their security properties. Despite its simplicity, the model captures the necessary features of distributed transaction systems and protocols. As part of this model, we define what it means for transactions to be serializable and what it means for a protocol to serialize transactions both correctly and securely.

2.3.1 State and Events

Similarly to Lamport [81], we define a *system state* to include a finite set of *events*, representing a history of the system up to a moment in time. An event (denoted e) is an atomic native action that takes place at a *location*, which can be thought of as a physical computer on the network. Some events may represent read operations ("the variable x had the value 3"), or write operations ("2 was written into the variable y"). In Figures 2.3 and 2.4, for example, events are represented as ovals, and correspond to lines of code.

Also part of the system state is a causal ordering on events. Like Lamport's causality [81], the ordering describes when one event e_1 causes another event e_2. In this case, we say e_1 *happens before* e_2, written as $e_1 \rightarrowtriangle e_2$. This relationship would hold if, for example, e_1 is the sending of a message, and e_2 its receipt. The ordering ($\rightarrowtriangle$) is a strict partial order: irreflexive, asymmetric, and transitive. Therefore, $e_1 \rightarrowtriangle e_2$ and $e_2 \rightarrowtriangle e_3$ together imply $e_1 \rightarrowtriangle e_3$.

The arrows in Figures 2.2 to 2.4 show happens-before relationships for the transactions involved.

2.3.2 Information Flow Lattice

We extend Lamport's model by assigning to each event e a *security label*, written $\ell(e)$, which defines the confidentiality and integrity requirements of the event. Events are the most fine-grained unit of information in our model, so there is no distinction between the confidentiality of an event's *occurrence* and that of its *contents*. Labels in our model are similar to high and low event sets [116, 35]. In Figures 2.3 and 2.4, two security labels, `High` and `Low` (`H` and `L` for short), are represented by the events' positions relative to the dashed line.

For generality, we assume that labels are drawn from a lattice [46], depicted in Fig. 2.5. Information is only permitted to flow upward in the lattice. We write "$\ell(e_1)$ is below $\ell(e_2)$" as $\ell(e_1) \sqsubseteq \ell(e_2)$, meaning it is secure for the information in e_1 to flow to e_2.

For instance, in Fig. 2.3, information should not flow from any events labeled `H` to any labeled `L`. Intuitively, we don't want secret information to determine any non-secret events, because unauthorized parties might learn something secret. However, information can flow in the reverse direction: reading the patient's address (labeled `L`) can affect Patsy's printout (labeled `H`): $\texttt{L} \sqsubseteq \texttt{H}$.

The join ($\sqcup$) of two labels represents their least upper bound: $\ell_1 \sqsubseteq (\ell_1 \sqcup \ell_2)$ and $\ell_2 \sqsubseteq (\ell_1 \sqcup \ell_2)$. The meet ($\sqcap$) of two labels represents their greatest lower bound: $(\ell_1 \sqcap \ell_2) \sqsubseteq \ell_1$ and $(\ell_1 \sqcap \ell_2) \sqsubseteq \ell_2$.

Like events, each location has a label, representing a limit on events with which that location can be trusted. No event should have more integrity than its location. Similarly, no event should be too secret for its location to know. Thus, in Fig. 2.5, only events *to the left of* a location's label (i.e., region C in the figure) may take place at that location.

For example, consider Gloria's payment event at CountriBank in the Rainforest example Fig. 2.1. This event ($\mathbf{r_5}$ in Fig. 2.2) represents money moving from Gloria's account to Outel's. The label ℓ of $\mathbf{r_5}$ should not have any more integrity than CountriBank itself, since the bank controls $\mathbf{r_5}$. Likewise, the bank knows about $\mathbf{r_5}$, so ℓ cannot be more confidential than the CountriBank's label. This would put ℓ to the *left* of the label representing CountriBank in the lattice of Fig. 2.5.

Our prototype implementation of secure transactions is built using the Fabric system [91], so the lattice used in the implementation is based on the Decentralized Label Model (DLM) [105]. However, the results of this dissertation are independent of the lattice used.

2.3.3 Conflicts

Two events in different transactions may *conflict*. This is a property inherent to some pairs of events. Intuitively, conflicting events are events that must be ordered for data to be consistent. For example, if e_1 represents reading variable x, and e_2 represents writing x, then they conflict, and furthermore, the value read and the value written establish an ordering between the events. Likewise, if two events both write variable x, they conflict, and the system must decide their ordering because it affects future reads of x.

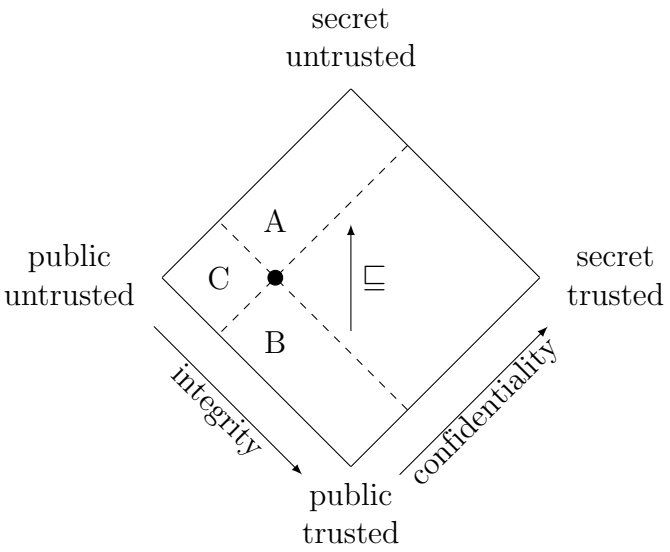

Figure 2.5: Security lattice: The dot represents a label in the lattice, and the dashed lines divide the lattice into four quadrants relative to this label. If the label represents an event, then only events with labels in quadrant B may influence this event, and this event may only influence events with labels in quadrant A. If the label represents a location, then only events with labels in quadrant C may occur at that location.

In our hospital example (Figures 2.3 and 2.4), the events Read address and Update address conflict. Specifically, the value read will change depending on whether it is read before or after the update. Thus for any such pair of events, there is a happens-before ($\twoheadrightarrow$) ordering between them, in one direction or the other.

We assume that conflicting events have the same label. This assumption is intuitive in the case of events that are reads and writes to the same variable (that is, storage location). Read and write operations in separate transactions could have occurred in either order, so the happens-before relationship between the read and write events cannot be predicted in advance.

Our notion of *conflict* is meant to describe direct interaction between transactions. Hence, we also assume any conflicting events happen at the same location.

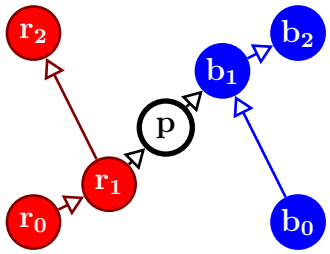

Figure 2.6: An example system state. The events $\mathbf{r_0}, \mathbf{r_1}$, and $\mathbf{r_2}$ form transaction $\mathbf{R}$, and the events $\mathbf{b_0}, \mathbf{b_1}$, and $\mathbf{b_2}$ form transaction $\mathbf{B}$. Event $\mathbf{p}$ is not part of either transaction. It may be an input, such as a network delay event, or part of a protocol used to schedule the transactions. In this state, $\mathbf{r_1} \rightarrow \mathbf{p} \rightarrow \mathbf{b_1}$, which means that $\mathbf{r_1}$ happens before $\mathbf{b_1}$, and so the transactions are ordered: $\mathbf{R} \rightarrow \mathbf{B}$.

2.3.4 Serializability and Secure Information Flow

Traditionally a transaction is modeled as a set of reads and writes to different objects [107]. We take a more abstract view, and model a transaction as a set of events that arise from running a piece of code. Each transaction features a *start event*, representing the decision to execute the transaction's code. Start events, by definition, happen before all others in the transaction. Multiple possible transactions can feature the same start event: the complete behavior of the transaction's code is not always determined when it starts executing, and may depend on past system events.

Fig. 2.4c shows two possible transactions, in blue, that can result from running the secure version of Patsy's code. They share the three events in solid blue, including the start event (Patsy start); one transaction contains a fourth event, Print address. The figure also shows in red the transaction resulting from Mallory's code. Fig. 2.6 is a more abstract example, in which $\mathbf{r_0}$ is the start event of transaction $\mathbf{R}$, and $\mathbf{b_0}$ is the start event of transaction $\mathbf{B}$.

In order to discuss what it means to *serialize* transactions, we need a notion of the *order* in which transactions happen. We obtain this ordering by lifting the happens-before relation on events to a happens-before ($\rightarrowtriangle$) relation for transactions. We say that transaction T_2 *directly depends* on T_1, written $T_1 \prec T_2$, if an event in T_1 happens before an event in T_2:

$$T_1 \prec T_2 \quad \equiv \quad T_1 \neq T_2 \wedge \exists e_1 \in T_1, e_2 \in T_2 \;.\; e_1 \rightarrowtriangle e_2$$

The happens-before relation on transactions ($\rightarrowtriangle$) is the transitive closure of this direct dependence relation $\prec$. Thus, in Fig. 2.6, the ordering **R**$\rightarrowtriangle$**B** holds. Likewise, Fig. 2.7 is a system state featuring the transactions from our hospital example (Fig. 2.4), in which Patsy$\rightarrowtriangle$Mallory holds.

Def. 1 (Serializability)**.** *Transactions are serializable exactly when happens-before is a strict partial order on transactions.*

Any total order consistent with this strict partial order would then respect the happens-before ordering ($\rightarrowtriangle$) of events. Such a total ordering would represent a *serial order* of transactions.

Def. 2 (Secure Information Flow)**.** *A transaction is* information-flow secure *if happens-before* ($\rightarrowtriangle$) *relationships between transaction events—and therefore causality—are consistent with permitted information flow:*

$$e_1 \rightarrowtriangle e_2 \quad \Longrightarrow \quad \ell(e_1) \sqsubseteq \ell(e_2)$$

This definition represents traditional information flow control within each transaction. Intuitively, each transaction itself cannot cause a security breach (although this definition says nothing about the protocol scheduling them). In our hospital example, Patsy's transaction in Fig. 2.3c is not *information-flow secure*, since Read

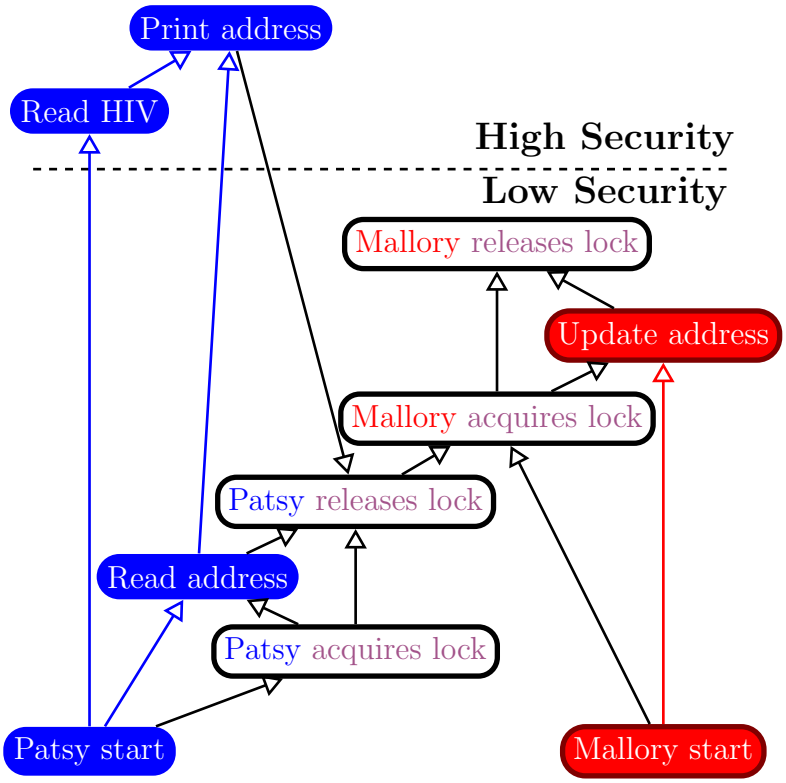

Figure 2.7: A possible system state after running transactions from Fig. 2.4c, assuming the patient has HIV, and an exclusive lock is used to order the transactions. (Events prior to everything in both transactions are not shown.) Because Patsy acquires the lock first, the transactions are ordered Patsy⇾Mallory. While each transaction is information-flow secure (a property of events within a transaction), when Patsy releases the lock after her transaction, a high security event happens before a low security one. We discuss secure scheduling protocols in § 2.6.

HIV happens before Read address, and yet the label of Read HIV, `H`, does not flow to the label of Read address, `L`. However, in the modified, secure version (Fig. 2.4c), there are no such insecure happens-before relationships, so Patsy's transaction is secure.

2.3.5 Network and Timing

Although this model abstracts over networks and messaging, we consider a message to comprise both a *send event* and a *receive event*. We assume asynchronous

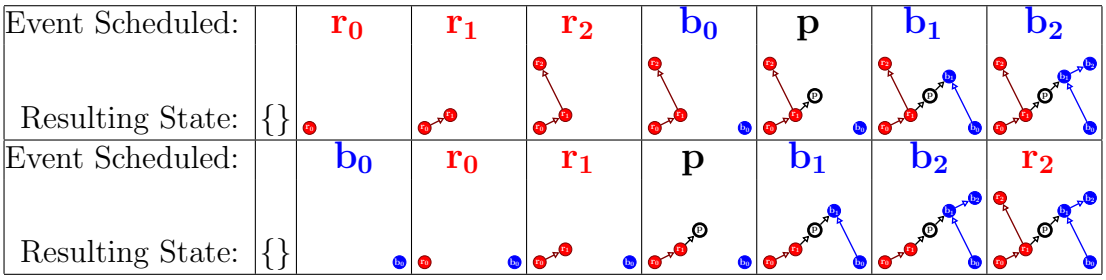

Figure 2.8: Two equivalent full executions for the system state from Fig. 2.6. Each begins with a start state (the empty set for full executions), followed by a sequence of events, each of which corresponds to the resulting system state.

---

messaging: no guarantees can be made about network delay. Perhaps because this popular assumption is so daunting, many security researchers ignore timing-based attacks. There are methods for mitigating leakage via timing channels [78, 10, 15] but in this work we too ignore timing.

To model nondeterministic message delay, we introduce a *network delay event* for each message receipt event, with the same label and location. The network delay event may occur at any time after the message send event. It must happen before ($\rightarrowtriangle$) the corresponding receipt event. In Fig. 2.6, event $\mathbf{r_1}$ could represent sending a message, event **p** could be the corresponding network delay event, which is not part of any transaction, and event $\mathbf{b_1}$ could be the message receipt event. Fig. 2.6 does not require **p** to be a network delay event. It could be any event that is not part of either transaction. For example, it might be part of some scheduling protocol.

2.3.6 Executions, Protocols, and Inputs

An *execution* is a start state paired with a totally ordered sequence of events that occur after the start state. This sequence must be consistent with happens-before

($\rightarrowtriangle$). Recall that a system state is a set of events (§ 2.3.1). Each event in the sequence therefore corresponds to a system state containing all the events in the start state, and all events up to and including this event in the sequence. Viewing an execution as a sequence of system states, an event is *scheduled* if it is in a state, and once it is scheduled, it will be scheduled in all later states. Two executions are *equivalent* if their start states are equal, and their sequences contain the same set of events, so they finish with equal system states (same set of events, same $\rightarrowtriangle$). A *full execution* represents the entire lifetime of the system, so its start state contains no events.

For example, Fig. 2.8 illustrates two equivalent full executions ending in the system state from Fig. 2.6.

A transaction scheduling *protocol* determines the order in which each location schedules the events of transactions. Given a set of possible transactions, a location, and a set of events representing a system state at that location, a protocol decides which event is scheduled next by the location:

$$\mathit{protocol} : \mathbf{set}\ \langle \text{Transactions} \rangle \times \text{Location} \times \text{State} \rightarrow \text{event}$$

Protocols can schedule an event from a started (but unfinished) transaction, or other events used by the protocol itself. In order to schedule transaction events in ways that satisfy certain constraints, like serializability, protocols may have to schedule additional events, which are not part of any transaction. These can include message send and receipt events. For example, in Fig. 2.7, the locking events are not part of any transaction, but are scheduled by the protocol in order to ensure serializability.

Certain kinds of events are not scheduled by protocols, because they are not under the control of the system. Events representing external inputs, including the start events of transactions, can happen at any time: they are fundamentally nondeterministic. We also treat the receive times of messages as external inputs. Each message receive event is the deterministic result of its send event and of a nondeterministic *network delay event* featuring the same security label as the receive event. We refer to start and network delay events collectively as *nondeterministic input events* (NIEs).

Protocols do not output NIEs. Instead, an NIE may appear at any point in an execution, and any prior events in the execution can happen before ($\twoheadrightarrow$) the NIE. Recall that an execution features a sequence of events, each of which can be seen as a system state featuring all events up to that point. An execution E is consistent with a protocol p if every event in the sequence is either an NIE, or the result of p applied to the previous state at the event's location. We sometimes say *p results in E* to mean "E is consistent with p."

As an example, assume all events in Fig. 2.6 have the same location L, and no messages are involved. Start events $\mathbf{r_0}$ and $\mathbf{b_0}$ are NIEs. Every other event has been scheduled by a protocol. Fig. 2.8 shows two different executions, which may be using different protocols, determining which events to schedule in each state. We can see that in the top execution of Fig. 2.8, the protocol maps:

$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{\mathbf{r_0}\} \mapsto \mathbf{r_1}$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{\mathbf{r_0}, \mathbf{r_1}\} \mapsto \mathbf{r_2}$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{\mathbf{r_0}, \mathbf{r_1}, \mathbf{r_2}, \mathbf{b_0}\} \mapsto \mathbf{p}$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{\mathbf{r_0}, \mathbf{r_1}, \mathbf{r_2}, \mathbf{b_0}, \mathbf{p}\} \mapsto \mathbf{b_1}$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{\mathbf{r_0}, \mathbf{r_1}, \mathbf{r_2}, \mathbf{b_0}, \mathbf{p}, \mathbf{b_1}\} \mapsto \mathbf{b_2}$$

The protocol in the bottom execution of Fig. 2.8 maps:

$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{\mathbf{r_0}, \mathbf{b_0}\} \mapsto \mathbf{r_1}$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{\mathbf{r_0}, \mathbf{b_0}, \mathbf{r_1}\} \mapsto \mathbf{p}$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{\mathbf{r_0}, \mathbf{b_0}, \mathbf{r_1}, \mathbf{p}\} \mapsto \mathbf{b_1}$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{\mathbf{r_0}, \mathbf{b_0}, \mathbf{r_1}, \mathbf{p}, \mathbf{b_1}\} \mapsto \mathbf{b_2}$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{\mathbf{r_0}, \mathbf{b_0}, \mathbf{r_1}, \mathbf{p}, \mathbf{b_1}, \mathbf{b_2}\} \mapsto \mathbf{r_2}$$

Ultimately, a protocol must determine the ordering of transactions. If the exact set of start events to be scheduled (as opposed to start events possible) were always known in advance, scheduling would be trivial. A protocol should not require one transaction to run before another *a priori*: **start events from any subset of possible transactions may be scheduled at any time.** No protocol should result in a system state in which such a start event cannot be scheduled, or an incomplete transaction can never finish.

2.3.7 Semantic Security Properties

Consider an observer who can only "see" events at some security level ℓ or below. If two states S_1 and S_2 are indistinguishable to the observer, then after a program runs, noninterference requires that the resulting executions remain indistinguishable to the observer. Secret values, which the observer cannot see, may differ in S_1 and S_2, and may result in different states at the end of the executions, but the observer should not be able to see these differences.

Possibilistic Noninterference

David Sutherland's hyperproperty *Generalized Noninterference*[2] [142] generalizes Goguen and Meseguer's noninterference [61]. His model features "possible execution sequences", much like our *executions*, each of which is a sequence of system states. For a given observer, some information is *low observable,* meaning the observer may learn it. Other information is *high*, meaning it's too secret for the observer to know. His model also features some events, called "signals," representing *inputs*, which can be either low or high. Possibilistic Noninterference, then, requires that for any given execution E_1, it must be possible to change the high inputs of E_1 to those of any other valid execution E_2, and create a valid, *possible* execution E_3 without changing any low events:

$$\forall E_1, E_2. \exists E_3. \quad \begin{aligned} High\_inputs(E_3) &= High\_inputs(E_2) \wedge \\ Low\_events(E_3) &= Low\_events(E_1) \end{aligned}$$

In a sense, an observer can't make any observations that change the *possible* set of high inputs, but might be able to infer which are probable. This is recognized as a fairly weak form of noninterference in nondeterministic systems. [35]

In our hospital example, as illustrated in Fig. 2.4, the system determines which of Patsy's transactions will run based upon whether `p.hasHiv` is `true`. We can treat this condition to be a high-security event that happens before all reads of `p.hasHiv`. If we classify this past high-security event as input, and all low-security events as low-observable for Mallory, then we must ensure that when Patsy's code runs, the set of possible low-security events that result is the same regardless of whether `p.hasHiv`. Patsy's possible transactions in Fig. 2.4 ensure possibilistic

[2] McCullough coins the term "Generalized Noninterference" [98], and Clarkson and Schneider define hyperproperties [35].

noninterference, while her transactions in Fig. 2.3 do not, since whether or not Read address occurs depends on `p.hasHiv`.

Relaxed Observational Determinism

Semantic conditions for information security are typically based on some variant of noninterference [61, 118]. These variants are often distinguished by their approaches to nondeterminism. However, many of these semantic security conditions fail under *refinement*: if some nondeterministic choices are fixed, security is violated [153]. However, low-security observational determinism [116, 153] is a strong property that is secure under refinement: intuitively, if an observer with label ℓ cannot distinguish states S and S', that observer must not be able to distinguish any execution E beginning with S from *any* execution E' beginning with S':

$$(S \approx_\ell S') \Rightarrow E \approx_\ell E'$$

This property is *too* strong because it rules out two sources of nondeterminism that we want to allow: first, the ability of any transaction to start at any time, and second, network delays. Therefore, we relax observational determinism to permit certain nondeterminism. We only require that executions be indistinguishable to the observer if their NIEs are indistinguishable to the observer:

$$(S \approx_\ell S' \quad \wedge \quad \mathit{NIE}(E) \approx_\ell \mathit{NIE}(E')) \Rightarrow E \approx_\ell E'$$

We call this relaxed property *relaxed observational determinism*. It might appear to be equivalent to observational determinism, but with the NIEs encoded in the start states. This is not the case. If NIEs were encoded in the start states, protocols would be able to read which transactions will start and when messages will arrive in the future. Therefore relaxed observational determinism captures

something that observational determinism does not: unknowable but "allowed" nondeterminism at any point in an execution.

By deliberately classifying start events and network delays as input, we allow certain kinds of information leaks that observational determinism would not. Specifically, a malicious network could leak information by manipulating the order or timing of message delivery. However, such a network could by definition communicate information to its co-conspirators anyway. Information can also be leaked through the order or timing of start events. This problem is beyond the scope of this work.

Conditioning the premise of the security condition on the indistinguishability of information that is allowed to be released is an idea that has been used earlier [119], but not in this way, to our knowledge.

In our hospital example, as illustrated in Fig. 2.4, the system determines which of Patsy's transactions (the one with the dashed events, or the one without the dashed events) will run based on whether `p.hasHiv` is `true`. We can consider `p.hasHiv`'s value to be a high-security event that happens before all reads of `p.hasHiv`. If we classify this past high-security event as input, and all low-security events as low-observable for Mallory, then we must ensure that when Patsy's code runs, the low-security projections of resulting executions are always the same, regardless of whether `p.hasHiv`. Patsy's possible transactions in Fig. 2.4 allow for observational determinism, while her transactions in Fig. 2.3 do not, since whether or not Read address occurs depends on `p.hasHiv`. Whether or not the system actually maintains observational determinism, however, depends on the protocol scheduling the events.

Def. 3 (Protocol Security)**.** *A protocol is considered* secure *if the set of resulting executions satisfies relaxed observational determinism for any allowed sets of information-flow secure transactions and any possible NIEs.*

2.4 Impossibility

One of our contributions is to show that even in the absence of timing channels, there is a fundamental conflict between secure noninterference and serializability. Previous results showing such a conflict, for example the work of Smith et al. [135] consider only confidentiality and show only that timing channels are unavoidable.

Theorem 1 (Impossibility)**.** *No secure protocol*[3] *can serialize all possible sets of information-flow secure transactions.*[4]

We assume protocols cannot simply introduce an arbitrarily trusted third party; a protocol must be able to run using only the set of locations that have events being scheduled.

Proof. (by counterexample) Consider the counterexample shown in Fig. 2.9. Alice and Bob are both cloud computing providers who keep strict logs of the order in which various jobs start and stop. Highly trusted (possibly government) auditors may review these logs, and check for consistency, to ensure cloud providers are honest and fair. As competitors, Alice and Bob do not want each other to gain

[3] barring unforeseen cryptographic capabilities (§ 2.4.1)

[4] In fact, what we prove is stronger. Our proof holds for even possibilistic security conditions [98], which are weaker than relaxed observational determinism (see technical report [129]). No protocol whose resulting traces satisfy even this weaker condition can serialize all sets of information-flow secure transactions.

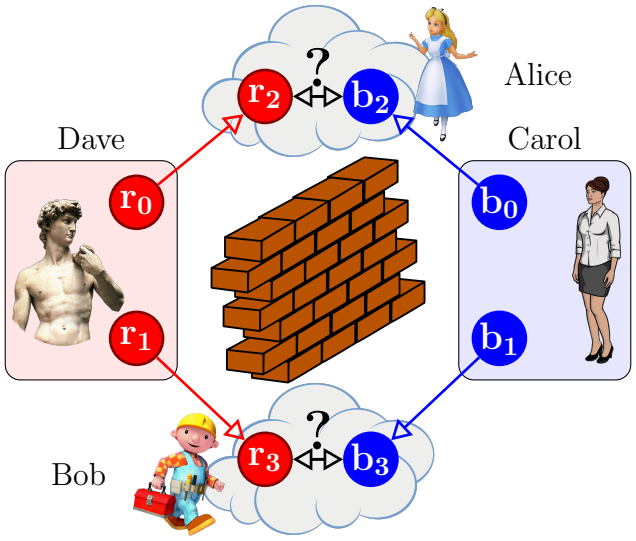

Figure 2.9: Transactions that cannot be securely serialized. Dave's transaction includes $\mathbf{r_0}$, $\mathbf{r_1}$, $\mathbf{r_2}$, and $\mathbf{r_3}$. Carol's includes $\mathbf{b_0}$, $\mathbf{b_1}$, $\mathbf{b_2}$, and $\mathbf{b_3}$. Cloud providers Alice and Bob must decide how to order their events. Alice and Bob may not influence each other, and Carol and Dave may not influence each other, as represented by the wall. For these transactions to be serializable, Alice's ordering of $\mathbf{r_2}$ and $\mathbf{b_2}$ must agree with Bob's ordering of $\mathbf{r_3}$ and $\mathbf{b_3}$.

---

any information about their services, and do not trust each other to affect their own services.

Carol and Dave are presently running jobs on Alice's cloud. Both Carol and Dave would like to stop their jobs on Alice's cloud, and start new ones on Bob's cloud. Each wants to do this atomically, effectively maintaining exactly one running job at all times. Carol and Dave consider their jobs to be somewhat confidential; they do not want each other to know about them. Unlike the example from Fig. 2.1, Dave and Carol's transactions do not go through a third party like Rainforest. For the transactions to be serializable, Alice's ordering of the old jobs stopping must agree with Bob's ordering of the new jobs starting.

These transactions feature at least 8 events:

$\mathbf{r_0}$: Dave sends a message to Alice

$\mathbf{r_1}$: Dave sends a message to Bob

$\mathbf{r_2}$: Alice receives a message from Dave, ending a job.

$\mathbf{r_3}$: Bob receives a message from Dave, beginning a job.

$\mathbf{b_0}$: Carol sends a message to Alice

$\mathbf{b_1}$: Carol sends a message to Bob

$\mathbf{b_2}$: Alice receives a message from Carol, ending a job.

$\mathbf{b_3}$: Bob receives a message from Carol, beginning a job.

No events at Alice's location should influence events at Bob's location, and vice-versa. No events at Carol's location should influence events at Dave's location, and vice-versa.

Alice and Bob must each finish with ordered logs including job beginnings and endings. This means they must assign a happens-before ($\rightarrowtriangle$) relation to their events above. For these transactions to be serializable, Alice's ordering of $\mathbf{r_2}$ and $\mathbf{b_2}$ must agree with Bob's ordering of $\mathbf{r_3}$ and $\mathbf{b_3}$.

Lemma 1. *These transactions are information-flow secure.*

The two transactions in Fig. 2.9 are information-flow secure (Def. 2).

Proof. The only happens-before relationships within transactions are for the sending and receipt of messages, explicitly carrying information readable to the recipient. All four are consistent with permitted information flows. □

Lemma 2. *No protocol can securely serialize these transactions. Specifically, no protocol accepting these transactions can preserve possibilistic noninterference.*

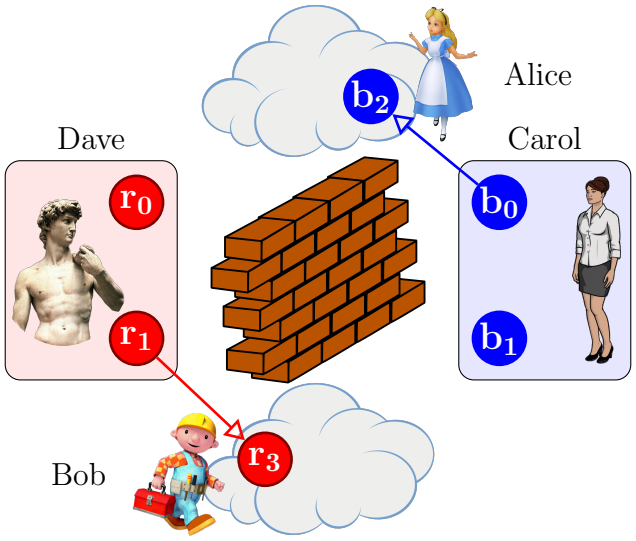

Figure 2.10: An intermediate state of an execution featuring the transactions from Fig. 2.9.

Proof. In any system with an asynchronous network, it is possible to reach a state in which Carol's message to Alice has arrived, but not her message to Bob, and Dave's message to Bob has arrived, but not his message to Alice. In other words, events $\mathbf{r_2}$ and $\mathbf{b_3}$ have not yet occurred. Fig. 2.10 illustrates this situation. In this state, neither Alice nor Bob can know whether one or both transactions have begun. It is impossible for either to communicate this information to the other without violating possibilistic noninterference. Specifically, any protocol that relayed such information from one cloud provider to the other would allow the recipient to distinguish the order of message delivery to the other cloud provider. That ordering is considered secret input, and so this would be a security violation. All executions with identical start states, and identical inputs visible to Alice, but differently ordered network delay events at Bob, which are inputs invisible to Alice, would become distinguishable to Alice. Even possibilistic noninterference would therefore be violated (§ 2.3.7).

Additionally, we have assumed that a protocol must be able to schedule any subset of the allowed transactions' start events. Therefore valid executions exist in which, say, only Carol's transaction runs, so Alice receives only information about Carol's transaction, and commits Carol's transaction first. Therefore a valid execution must exist in which Alice commits Carol's transaction first, before receiving any further input from Dave or Bob, and likewise, Bob commits Dave's transaction first, without further input from Carol or Alice. Thus any protocol satisfying possibilistic noninterference can schedule inconsistently: the transactions cannot be securely serialized. □

Thus, with this scenario as a counterexample, no secure protocol can serialize all possible sets of information-flow secure transactions. □

2.4.1 Cryptography

This essentially information-theoretic argument does not account for the possibility that some protocol could produce *computationally indistinguishable* traces that are low-distinguishable with sufficient computational power (e.g., to break encryption). However, we are unaware of any cryptographic protocols that would permit Alice and Bob to learn a consistent order in which to schedule events without learning each other's confidential information.

2.5 Analysis

Although secure scheduling is impossible in general, many sets of transactions can be scheduled securely. We therefore investigate which conditions are sufficient for secure scheduling, and what protocols can function securely under these conditions.

2.5.1 Monotonicity

A relatively simple condition suffices to guarantee schedulability, while preserving relaxed observational determinism:

Def. 4 (Monotonicity)**.** *A transaction is* monotonic *if it is information-flow secure and its events are totally ordered by happens-before* ($\rightarrowtriangle$)*.*

Theorem 2 (Monotonicity $\Rightarrow$ Schedulability)**.**
A protocol exists that can serialize any set of monotonic transactions and preserve relaxed observational determinism.

Proof. Monotonicity requires that each event must be allowed to influence all future events in the transaction. A simple, pessimistic transaction protocol can schedule such transactions securely. In order to define this protocol, we need a notion of *locks* within our model.

Locks. In distributed systems, a *lock* is an abstract token, that only one entity at a time can posses. Locks are used, for instance, to ensure two different programs don't try to use the same resource, even physical machinery, at the same time [132].

In our system model, a lock consists of an infinite set of events for each allowed transaction. A transaction *acquires* a lock by scheduling any event from this set.

It *releases* a lock by scheduling another event from this set. Thus, in a system state S, a transaction T *holds* a lock if S contains an odd number of events from the lock's set corresponding to T. No correct protocol should result in a state in which multiple transactions hold the same lock. All pairs of events in a lock conflict, so scheduled events that are part of the same lock must be totally ordered by happens-before ($\twoheadrightarrow$). All events in a lock share a location, which is considered to be the location of the lock itself. Likewise, all events in a lock share a label, which is considered to be the label of the lock itself.

A critical property for transaction scheduling is *deadlock freedom* [52, 132], which requires that a protocol can eventually schedule all events from any transaction whose start event has been scheduled. A system enters *deadlock* when it reaches a state after which this is not the case. For example, deadlock happens if a protocol requires two transactions each to wait until the other completes: both will wait forever. If all transactions are finite sets of events (i.e., all transactions can terminate), then deadlock freedom guarantees that a system with a finite set of start events eventually terminates, a liveness property. Deadlock freedom is essential to distributed or parallel scheduling, but notoriously difficult to get right [132].

We now describe a deadlock-free protocol that can securely serialize any set of monotonic transactions, and preserve relaxed observational determinism:

- Each event in each transaction has a corresponding lock, except start events.
- Any events that have the same label share a lock, and this lock shares a location with at least one of the events. Conflicting events are assumed to share a label (§ 2.3.4).
- A transaction must hold an event's lock to schedule that event.

- A transaction acquires locks in sequence, scheduling events as it goes. Since all events are ordered according to a global security lattice, all transactions that acquire the same locks do so in the same order. Therefore they do not deadlock.
- If a lock is already held, the transaction waits for it to be released.
- When all events are scheduled, the transaction commits, releasing locks in reverse order. Any messages sent as part of the transaction would thus receive a reply, indicating only that the message had been received, and all its repercussions committed. We call these replies *commit* messages.
- For each location, the protocol rotates between all uncommitted transactions, scheduling any intermediate events (such as lock acquisitions) until it either can schedule one event in the transaction or can make no progress, and then rotates to the next transaction.

Security Intuition. Acquiring locks shared by multiple events on different locations requires a commit protocol between those locations. However, this does not leak information because all locations involved are explicitly allowed to observe and influence all events involved. Therefore several known commit protocols will do, including 2PC. Since the only messages sent as part of the protocol are commit messages, and each recipient knows it will receive a commit message by virtue of sending a message in the protocol, no information (other than timing) is transferred by the scheduling mechanism itself.

Relaxed observational determinism. This protocol, implemented with monotonic transactions, satisfies relaxed observational determinism, our slightly relaxed version of observational determinism (§ 2.3.7). We consider an event *observable* to an observer with label ℓ if the label of the event flows to ℓ. For any

two executions beginning with equivalent states (for some observer ℓ),

$$E_0[0] \approx_\ell E_1[0]$$

If the executions E_0 and E_1 have the same ℓ-observable inputs, which is to say transaction start events and network delay events, then the protocol requires E_0 and E_1 to be indistinguishable to ℓ. The observer of label ℓ can only observe a prefix of each transaction being scheduled in a round-robin fashion, and commit messages for each arriving sometime thereafter. Arrival time of these commit messages is considered an input, and so all events visible in E_0 and E_1 are deterministic results of the events visible in the start states, and the NIEs. Each distinct state in an execution, as observed at ℓ, will be deterministically predicted by prior states and inputs. Thus relaxed observational determinism is preserved.

Serializability. Transactions consist of totally ordered series of events. Let e_1 be the first event in T_1 conflicting with any event in T_2. Let e_2 be the event in T_2 with which e_1 conflicts. Suppose they are scheduled such that $e_1 \rightarrowtriangle e_2$. Therefore all events in T_2 after and including e_2 cannot be scheduled until T_1 commits and releases its locks. No event in T_2 scheduled before e_2 can conflict with an event in T_1 after e_1, by monotonicity, or before e_1, by the definition of e_1. Thus all conflicting events in T_2 are scheduled after all events in T_1, so no event in T_1 can happen after an event in T_2. Therefore, this pessimistic protocol ensures serializability.

Liveness. This scheduling system cannot result in deadlock, since all transactions acquire locks in strictly increasing order on the lattice, so any set of transactions that acquire the same locks must do so in the same order.

Therefore, monotonicity is sufficient to guarantee secure schedulability. □

2.5.2 Relaxed Monotonicity

Monotonicity, while relatively easy to understand, is not the weakest condition we know to be sufficient for secure schedulability. It can be substantially relaxed. In order to explain our weaker condition, *relaxed monotonicity*, we first need to introduce a concept we call *visibility*:

Def. 5 (Visible-To)**.** *An event e in transaction T is* visible to *a location L if and only if it happens at L, or if there exists another event $e' \in T$ at L, such that $e \rightarrowtriangle e'$.*

Def. 6 (Relaxed Monotonicity)**.** *A transaction T satisfies* relaxed monotonicity *if it is information-flow secure and for each location L, all events in T visible to L happen before all events in T not visible to L.*

In § 2.6, we demonstrate that relaxed monotonicity guarantees schedulability. Specifically, we present a staged commit protocol, and prove that it schedules any set of transactions satisfying relaxed monotonicity, while preserving relaxed observational determinism (Thm. 4).

2.5.3 Requirements for Secure Atomicity

Monotonicity and relaxed monotonicity are sufficient conditions for a set of transactions to be securely schedulable. Some sets of transactions meet neither condition, but can be securely serialized by some protocol. For example, any set of transactions that each happen entirely at one location can be securely serialized if each location schedules each transaction completely before beginning the next. We now describe a relatively simple condition that is necessary for any set of transactions to be securely scheduled.

Decision Events and Conflicting Events

In order to understand this necessary condition, we first describe *decision events* and *conflicting events*.

Borrowing some terminology from Fischer, Lynch, and Paterson [54], for a pair of transactions T_1 and T_2, any system state is either *bivalent* or *univalent*. A system state is *bivalent* with respect to T_1 and T_2 if there exist two valid executions that both include that state, but end with opposite orderings of T_1 and T_2. A system state is *univalent* with respect to T_1 and T_2 otherwise: for one ordering of the transactions, no valid execution ending with that ordering contains the state.

We can define a similar relationship for start events: for any pair of distinct start events s_1 and s_2, a system state is bivalent with respect to those events if it features in two valid executions, both of which have s_1 and s_2 in scheduled transactions, but those transactions are in opposite order. A system state is univalent with respect to s_1 and s_2 otherwise.

All full executions (i.e., those starting with an empty state) that order a pair of transactions begin in a *bivalent* state with respect to their start events, before either is scheduled. By our definition of serializability and transaction ordering, once transactions are ordered, they cannot be un-ordered. Any execution that orders the transactions therefore ends in a *univalent* state with respect to their start events. Any such execution consists of a sequence of 0 or more *bivalent* states followed by a sequence of *univalent* states. The event that is scheduled in the first *univalent* state, in a sense, decides the ordering of the transactions. We call it the *decision event*.

We call any event in T_1 or T_2 that conflicts with an event in the other transaction a *conflicting event*.

Lemma 3 (Decision Event $\rightarrow\!\!\triangleright$ Conflicting Events)**.**
For any univalent state S with $T_1 \rightarrow\!\!\triangleright T_2$, there exists a full execution E ending in S featuring a decision event e_d that happens before ($\rightarrow\!\!\triangleright$) all conflicting events in T_1 and T_2 (other than e_d itself, if e_d is a conflicting event).

Proof. Assume the contradiction. Then for any full execution E' ending in S, an equivalent execution exists featuring a state in which a conflicting event e_c is scheduled, but the decision event of E' is not. Such an equivalent execution would by definition have a different decision event, since e_c's presence in a state makes the state univalent. By our assumption, this equivalent execution has conflicting events that neither are, nor happen after, its decision event. This implies yet another equivalent execution with yet another state featuring an even earlier conflicting event but not the decision event, and so on. Since all states are finite sets, and $\rightarrow\!\!\triangleright$is a strict partial order, this infinite descending chain is impossible. There must exist an execution E ending with S with decision event e_d that happens before all conflicting events in T_1 and T_2. □

We show that two fundamental system state properties are necessary for secure scheduling:

Def. 7 (First-Precedes-Decision)**.** *State S satisfies* First-Precedes-Decision *if, for any pair of transactions T_1 and T_2 in S with $T_1 \rightarrow\!\!\triangleright T_2$, there is a full execution E ending in S with a decision event e_d that either is in T_1, or happens after an event in T_1.*

Def. 8 (Decision-Precedes-Second)**.** *A state S satisfies* Decision-Precedes-Second *if, for any pair of transactions T_1 and T_2 in S with $T_1 \twoheadrightarrow T_2$, there is a full execution E' ending in S with a decision event e'_d, such that no event in T_2 happens before e'_d.*

Therefore, for a protocol to be secure, it must ensure resulting system states have these properties.

Theorem 3 (Necessary Condition)**.** *Any secure, deadlock-free protocol p must ensure that all full executions consistent with p feature only states satisfying both First-Precedes-Decision and Decision-Precedes-Second.*

Proof. Given $T_1 \twoheadrightarrow T_2$, any execution E' ending in S features a decision event e_d. Decision events for the same pair of transactions in equivalent executions must agree on ordering, by the definition of equivalent execution. If T_1 does not contain E's decision event, e_d, or any event that happens before e_d, then there exists an equivalent execution in which e_d is scheduled before any events in T_1 or T_2. This execution would imply the existence of a system state in which no event in either transaction is scheduled, but it is impossible to schedule T_2 before T_1, regardless of inputs after that state. If, after this state, the start event for T_2 were scheduled, but not the start event for T_1, then T_2 cannot be scheduled. This contradicts a the deadlock-freedom requirement: no protocol should result in a system state in which a supported transaction can never be scheduled.

Therefore some event in T_1 either is or happens before e_d for some full execution E ending in S.

If T_1 and T_2 conflict, then e'_d either is an event in T_1 or happens before an event in T_1, by Lemma 3. If an event $e_2 \in T_2$ happens before e'_d, then either $e'_d \in T_1$, and

$$e_2 \rightarrow e'_d \Rightarrow T_2 \rightarrow T_1$$

which is impossible, by the definition of happens-before, or
$\exists e_1 \in T_1 . e'_d \rightarrow e_1$, and

$$e_2 \rightarrow e'_d \rightarrow e_1 \Rightarrow e_2 \rightarrow e_1 \Rightarrow T_2 \rightarrow T_1$$

which is also impossible, by the definition of happens-before.

If T_1 and T_2 do not conflict, then the only way $T_1 \rightarrow T_2$ implies that there exists some chain
$T_1 \rightarrow T_3 \rightarrow T_4 \rightarrow \ldots \rightarrow T_n \rightarrow T_2$ such that and each transaction in the chain conflicts with the next. Therefore, by the above proof, an equivalent execution exists in which each transaction in the chain contains the decision event for ordering itself and the following transaction, and no events in the following transaction are before that decision event.

Therefore there exists some equivalent execution E' in which no event in T_2 happens before the decision event e'_d deciding the ordering between T_1 and T_2. $\square$

Although Thm. 3 may seem trivial, it represents some important conclusions: No protocol can make any final ordering decision until at least one transaction involved has begun. Furthermore, it is impossible for the later transaction to determine the decision. Truly atomic transactions cannot include any kind of two-way interaction or negotiation for scheduling.

2.6 The Staged Commit Protocol

We now present the staged commit protocol (SC) and prove that it is secure, given transactions satisfying relaxed monotonicity.

SC is a hybrid of traditional serialization protocols, such as 2PC, and the simple pessimistic protocol described in the proof of Thm. 2. Compared to our simple pessimistic protocol, it allows a broader variety of transactions to be scheduled (relaxed monotonicity vs. regular monotonicity), which in turn allows more concurrency. A transaction is divided into *stages*, each of which can be securely committed using a more traditional protocol. The stages themselves are executed in a pessimistic sequence.

Each event scheduled is considered to be either *precommitted* or *committed*. We express this in our model by the presence or absence of an "isCommitted" event corresponding to every event in a transaction. Intuitively, a *precommitted* event is part of some ongoing transaction, so no conflicting events that happen after a precommitted event should be scheduled. A *committed* event, on the other hand, is part of a completed transaction; conflicting events that happen after a committed event can safely be scheduled. Once an event is precommitted, it can never be un-scheduled. It can only change to being committed. Once an event is committed, it can never change back to being precommitted.

- The events of each transaction are divided into stages. Each stage will be scheduled using traditional 2PC, so aborts within a stage will be sent to all locations involved in that stage.

  To divide the events into stages, we establish equivalence classes of the events' labels. Labels within each class are *equivalent* in the following sense: when

events with equivalent labels are aborted, those aborts can securely flow to the same set of locations. An event's abort can always flow to the event's own location, so locations involved in a stage can securely ensure the atomicity of the events in that stage. Since conflicting events have the same security labels, they will be in the same equivalence class. We call these equivalence classes *conflict labels* (`cl`).

- Each stage features events of the same conflict label, and is scheduled with 2PC. One location must coordinate the 2PC. All potential aborts in the stage must flow to the coordinator, and some events on the coordinator must be permitted to affect all events in the stage. Relaxed monotonicity implies that at least one such location exists for each conflict label.

  When a stage tries to schedule an event, but finds a precommitted conflicting event, it aborts the entire stage. Because conflicting events have the same label, these aborts cannot affect events on unpermitted locations.

  When a stage's 2PC completes, the events in the stage are scheduled, and considered *precommitted*.

- Each transaction precommits its stages as they occur. To avoid deadlock, we must ensure that whenever two transactions feature stages with equal conflict labels, they precommit those stages in the same order. Therefore, the staged commit protocol assumes an ordering of conflict labels. This can be any arbitrary ordering, so long as (1) it totally orders the conflict labels appearing in each transaction, and (2) all transactions agree on the ordering.

- When all stages are precommitted, all events in the transaction can be committed. *Commit* messages to this effect are sent between locations, backwards through the stages. Whenever an event in one stage triggers an event

in the next, the locations involved can be sure a *commit* message will take the reverse path. The only information conveyed is timing.

Because events in a precommitted stage cannot be un-scheduled or "rolled back", a participant that is involved only in an earlier stage is prevented from gleaning any information about later stages. The participant will only learn, eventually, that it can commit.

Patsy's transaction in Fig. 2.4c has at least two stages when the patient has HIV:

1. Patsy begins the transaction (Patsy start), and reads the address (Read Address). This stage will be atomically precommitted, and this precommit process will determine the relative ordering of Patsy's transaction and Mallory's, independent of more secret events.
2. Patsy finds that the patient has HIV (Read HIV), and prints the patient's address (Print address).

Theorem 4 (Security of SC)**.** *Any set of transactions satisfying relaxed monotonicity are serialized by SC securely without deadlock.*

Proof.

Security. SC preserves relaxed observational determinism. Intuitively, any information flows that it adds are already included in the transaction.

SC adds no communication affecting security:

- Communication within each stage is strictly about events that all participants can both observe.

- For each pair of consecutive stages, at least one participant from the first stage can notify a participant in the second stage securely, when it is time for the second stage to begin. Relaxed monotonicity ensures the second stage contains an event that happens after an event in the first stage, representing a line of communication.
- Communication for commits can safely proceed in reverse order of stages. Within each stage, each participant can securely forward a commit message to all other participants. Between stages, commit messages can be sent back along the same channels used to notify each stage the previous one had precommitted. Each participant knows when it precommits exactly which commit messages it will receive. The commit messages themselves do not leak any information (other than timing) to their recipients.

Therefore SC adds no unauthorized information flows.

Specifically, for any given participant's label ℓ, events within a stage visible to ℓ are scheduled deterministically based only on information visible to ℓ. Commit messages (and affiliated events) for visible stages arrive eventually, at a time determined by network delay events, which we consider input. Other stages' events are not observable to ℓ.

Therefore, for any two executions beginning with states indistinguishable to ℓ, with NIEs visible to ℓ, all scheduled events visible to ℓ would be indistinguishable. Thus relaxed observational determinism is preserved.

Serializability. Any set of transactions with relaxed monotonicity scheduled by SC will be serializable.

Lemma 4 (Precommitted Snapshot)**.**

Any execution in which an event in a transaction is committed features a system state in which all events in the transaction are precommitted.

Proof. Stages are totally ordered, and each waits until the final stage commits before ($\rightarrowtriangle$) any of its events commit. The final stage precommits before ($\rightarrowtriangle$) it commits, and so there is a system state in which all events in the transaction are precommitted. □

Let E be an execution where any two conflicting transactions T_1 and T_2 both have at least one event that commits. Given Lemma 4, E must feature two states: one in which all events in T_1 are precommitted, and another in which all events of T_2 are precommitted. As T_1 and T_2 conflict, these states cannot be the identical. (An event is never scheduled while a conflicting event is precommitted.)

One transaction must be scheduled before ($\rightarrowtriangle$) the other. Without loss of generality, let it be T_1. No equivalent execution can feature a state in which an event in T_2 is scheduled before an event in T_1, as this would require a conflicting event in T_2 to be precommitted before its corresponding conflicting event in T_1 is committed. The corresponding conflicting event in T_1 must be precommitted before any event in T_1 commits, and we require that all events in q_2 remain precommitted until after an event in T_1 commits.

Therefore, if $T_1 \rightarrowtriangle T_2$ then it is impossible for $T_2 \rightarrowtriangle T_1$. Thus SC guarantees a strict partial order of transactions, and therefore serializability.

Deadlock Freedom.

A deadlock can occur only if there is a cycle of dependencies among transactions, in which transaction T_1 *depends on* T_2 if and only if T_2 has precommitted an event conflicting with an unscheduled event in T_1.

Conflicting events share labels, and stages are defined by labels. All transactions must therefore order the stages of conflicting pairs in the same way. One event can only ever depend on an event in its own or in a prior stage. Stages are precommitted in order, so no dependency cycle featuring events in different stages is possible.

Each stage is precommitted atomically using 2PC. 2PC preserves deadlock freedom, meaning no cycle featuring only events in the same stage is possible.

Therefore no cycles, and thus no deadlock, can exist with SC.

SC is secure, deadlock-free, and guarantees serializability when the transactions have relaxed monotonicity. □

The Importance of Optimism

SC specifies only a commit protocol. Actual computation (which generates the set of events) for each transaction can be done in advance, optimistically. If one stage precommits and the next is blocked by a conflicting transaction, optimistically precomputed events would have to be *rolled back*. However, no precommitted event need be rolled back. In fact, it would be insecure to do so. Thus SC allows for partially optimistic transactions with partial rollback.

Our model requires only that a transaction be a set of events. In many cases, however, it is not possible to know which transaction will run when a start event is

scheduled. For example, a transaction might read a customer's banking information from a database and contact the appropriate bank. It would not be possible to know which bank should have an event in the transaction beforehand. If a system attempted to read the banking information prior to the transaction, then serializability is lost: the customer might change banks in between the read and the transaction, and so one might contact the wrong bank.

Optimism solves this problem: events are precomputed, and when an entire stage is completed, that stage's 2PC begins. This means that optimism is not just an optimization; it is required for secure scheduling in cases where the transactions' events are not known in advance.

2.7 Implementation

We extended the Fabric language and compiler to check that transactions can be securely scheduled, and we extended the Fabric runtime system to use SC. Fabric and IFDB [122] are the two open-source systems we are aware of that support distributed transactions on persistent, labeled data with information flow control. Of these, we chose Fabric for its static reasoning capabilities. IFDB checks labels entirely dynamically, so it cannot tell if a transaction is schedulable until after it has begun.

2.7.1 The Fabric Language

The Fabric language is designed for writing distributed programs using atomic transactions that operate on persistent, Java-like objects [91]. It has types that la-

bel each object field with information flow policies for confidentiality and integrity. The compiler uses these labels to check that Fabric programs enforce a noninterference property. However, like all modern systems built using 2PC, Fabric does not require that transactions be securely scheduled according to the policies in the program. Consequently, until now, abort channels have existed in Fabric.

We leverage these security labels and extend the compiler to additionally check that transactions in a Fabric program are monotonic (§ 2.5). This implementation prevents confidentiality breaches via abort channels. Preventing integrity breaches would require further dynamic checks, which we leave to future work.

2.7.2 Checking Monotonicity

Our modification to the Fabric compiler enforces relaxed monotonicity (Def. 6). Our evaluation (§ 2.8) shows that enforcing this condition does not exclude realistic and desirable programs. Our changes to the Fabric compiler and related files include 4.1k lines of code (out of roughly 59k lines).

Events and Conflict Labels in Fabric

The events in the system model (§ 2.3) are represented in our implementation by read and writes on fields of persistent Fabric objects. The label of the field being read or written corresponds to the event labels in our model.

SC (§ 2.6) divides events into stages based on conflict labels (`cl`). In our implementation, we define the `cl` of an event e to correspond to the set of *principals* authorized to read or write the field that is being accessed by e. If e is a write event,

```
1 atomic {                          PC Possible conflictors
2   String{ℓ} p = post.read();      ⊥  {Alice, Bob, Carol}
3   Comments{ℓ′} c;                 ⊥    -
4   if (p.contains("fizz")) {       ⊥    -
5     c.write("buzz");              ℓ    {Alice, Carol}
6   if (p.contains("buzz")) {       ⊥    -
7     c.write("fizz");              ℓ    {Alice, Carol}
8   }
9 }
```

Figure 2.11: Carol's program in our Blog example: Carol reads a post with label ℓ, and depending on what she reads, writes a comment with label ℓ'. Label ℓ permits Alice, Bob, and Carol to read the post, while ℓ' keeps the Comments more private and allows only Alice and Carol to view or edit.

---

this set contains exactly those principals that can perform a conflicting operation (and thereby receive an abort); if e is a read event, the set is a conservative over-approximation, since only the writers can conflict.

Fig. 2.11 presents a program in which Carol schedules two events within a single transaction. First, she reads a blog post with security label ℓ. Second, she writes a comment (whose content depends on that of the post) with label ℓ'. Since ℓ permits Alice, Bob, or Carol to read the post, the `cl` of the first event includes all three principals. However, only Alice and Carol can read or write the comment, so when Carol goes to write it, only Alice or another transaction acting on behalf of Carol could cause conflicts. The `cl` of the write therefore includes only Alice and Carol.

Program Counter Label

The *program counter* label (`pc`) [47] labels the program context. For any given point in the code, the `pc` represents the join (least upper bound) of the labels of events that determine whether or not execution reaches that point in the code.

These events include those occurring in if-statement and loop conditionals. For instance, in Fig. 2.11, whether line 5 runs depends on the value of `p`, which has label ℓ. Therefore, the fact that line 5 is executing is as secret as `p`, and the `pc` at line 5 is ℓ.

SC requires that when events with the same `cl` are aborted, those aborts can securely flow to the same set of locations. When an event causes an abort, the resulting abort messages carry information about the context in which the event occurs. Therefore, we enforce the requirement by introducing a constraint on the program context in which events may occur: the `pc` must flow to the principals in the conflict label.

$$\texttt{pc} \sqsubseteq \texttt{cl} \tag{2.1}$$

Eliding the details of how Fabric's labels are structured, in Fig. 2.11, $\bot$ flows to everything, and ℓ, the label of the blog post, does flow to the conflict label, indicating that both Alice and Carol can cause a conflict. Therefore, Eqn. (2.1) holds on lines 2, 5, and 7.

Ordering Stages

Each stage consists of operations with the same `cl`. To ensure all transactions precommit conflicting stages in the same order, we adopt a universal stage ordering:

$$\mathit{principals}(\texttt{cl}_i) \supsetneq \mathit{principals}(\texttt{cl}_{i+1}) \tag{2.2}$$

The set of principals in each stage must be a strict superset of the principals in the next one. This ensures that unrestricted information can be read in one stage and sensitive information can be modified in a later stage in the same transaction. In the hospital example (Fig. 2.4), Read HIV has a conflict label that only includes

trusted personnel, while Read address has a conflict label that includes more hospital staff. As a result, our implementation requires that Read address be staged before Read HIV in Patsy's transaction.

In Fig. 2.11, our stage ordering means that the read on line 2, with a `cl` of $\{Alice, Bob, Carol\}$ belongs in an earlier stage than the write, which features a `cl` of only $\{Alice, Carol\}$.

Method Annotations

To ensure modular program analysis and compilation, each method is analyzed independently. Fabric is an object-oriented language with dynamic dispatch, so it is not always possible to know in advance which method implementation a program will execute. Therefore, the exact conflict labels for events within a method call are not known at compile time. In order to ensure each atomic program can divide into monotonic stages, we annotate each method with bounds on the conflict labels of operations within the method. These annotations are the security analogue of argument and return types for methods.

2.7.3 Implementing SC

We extended the Fabric runtime system to use SC instead of traditional 2PC, modifying 2.4k lines of code out of a total of 24k lines of code in the original implementation. Specifically, we changed Fabric's 2PC-based transaction protocol so that it leaves each stage prepared until all stages are ready, and then commits.

Since Fabric labels can be dynamic, the compiler statically determines *potential stagepoints*—points in the program that may begin a new stage—along with the conflict labels of the stages immediately surrounding the potential stagepoint. If the compiler cannot statically determine whether the conflict labels before and after a stagepoint will be different, it inserts a dynamic equivalence check for the two labels. At run time, if the two labels are not equivalent, then a stage is ending, and the system precommits all operations made thus far. To precommit a stage, we run the first ("prepare") phase of 2PC. If there is an abort, the stage is re-executed until it eventually precommits.

In Fig. 2.11, there is a potential stagepoint before lines 4 and 6, where the next operation in each case will not include Bob as a possible conflictor. The conflict labels surrounding the potential stagepoint are $\{Alice, Bob, Carol\}$ (from reading the post on line 2) and $\{Alice, Carol\}$ (from writing the comment on either line 4 or 6). If another transaction caused the first stage to abort, then Carol's code would rerun up to line 4 or 6 until it could precommit, and then the remainder of the transaction would run.

2.8 Evaluation

To evaluate our implementation, we built three example Fabric applications, and tested them using our modified Fabric compiler:

- an implementation of the hospital example from section 2.2;
- a primitive blog application (from which Figure 2.11 was taken), in which participants write and comment on posts with privacy policies; and

- an implementation of the Rainforest example from section 2.2.

2.8.1 Hospital

We implemented the programs described in our hospital example (Figure 2.3). In the implementation, Patsy's code additionally appends the addresses of HIV-positive patients to a secure log. In a third program, another trusted participant reads the secure log.

With our changes, the compiler correctly rejects Patsy's code. We amended her code to reflect Figure 2.4. Of the 350 lines of code, we had to change a total of 113 to satisfy relaxed monotonicity and compile. Of these 113 lines, 23 were additional method annotations and the remaining 90 were the result of refactoring the transaction that retrieves the addresses of HIV-positive patients. SC scheduled the transactions without leaking information. The patient's HIV status made Mallory neither more nor less likely to receive aborts.

2.8.2 Blog

In our primitive blog application, a store holds API objects, each of which features blog posts (represented as strings) with some security label, and comments with another security label. These labels control who can view, edit, or add to the posts and comments.

In one of our programs, the blog owner atomically reads a post and updates its text to alternate between "fizz" and "buzz". In another program, another user comments on the first post (Figure 2.11). To keep this comment pertinent to the

Table 2.1: Example policies for the Rainforest application.

Data item	Readers	Writers
Gloria's account balance	Bank, Gloria	Bank
Item price	(public)	Outel
Inventory	Outel	Outel

content of the post, reading the post and adding the comment are done atomically. Since posts and comments have different labels, this transaction has at least two stages: one to read the post, and another to write the comment.

We were able to compile and run these programs with our modified system with relatively few changes. Of the 352 lines of code, we had to change a total of 50, primarily by adding annotations to method signatures (section 2.7.2).

2.8.3 Rainforest

We implemented the Rainforest example from subsection 2.2.1. In our code, two nodes within Rainforest act with Rainforest's authority. They perform transactions representing the orders of Gloria and Fred from Figure 2.1. Each transaction updates inventory data stored at one location, and banking data stored at another. Table 2.1 gives examples of the policies for price, inventory, and banking data.

While attempting to modify this code to work with SC, we discovered that the staging order chosen in section 2.7.2 makes it impossible to provide the atomicity of the original application while both meeting its security requirements and ensuring deadlock freedom.

To illustrate, suppose Gloria is purchasing an item from Outel. To ensure she is charged the correct price, the event that updates the inventory must share a

transaction with the one that debits Gloria's bank account. The conflict label for the inventory event corresponds to {`Outel`}, whereas the conflict label for the debit event corresponds to {`Bank`, `Gloria`}. Since neither is a subset of the other, the compiler cannot put them in the same transaction.

These difficulties in porting the Rainforest application arise because Fabric is designed to be an open system, and so an *a priori* choice of staging order must be chosen. If the application were written as part of a closed system, deadlock freedom can be achieved by picking a staging order that works for this particular application (e.g., {`Outel`} before {`Bank`, `Gloria`}), but it might be difficult to extend the system with future applications.

2.8.4 Overhead

The staged commit protocol adds two main sources of overhead compared to traditional 2PC. First, each stage involves a round trip to prepare the data manipulated during the stage, leading to overhead that scales with the number of stages and with network latency. Second, as described in subsection 2.7.3, dynamic labels result in potential stagepoints, which must be resolved using run-time checks. The number of checks performed depends on how well the compiler's static analysis predicts potential stagepoints.

We measured this overhead in our implementation on an Intel Core i7-2600 machine with 16 GiB of memory, using the transactions in our examples. The post and comment transactions in the blog example were each run continually for 15 minutes, and Patsy's transaction in the hospital example was run continually for 1 hour.

Table 2.2: Performance overhead of SC. Reported times are per-transaction averages, across three 5-minute runs of the blog application and three 20-minute runs of the hospital application. Relative standard error of all measurements is less than 2%.

Example	Program	SC			2PC
		# stages	Dyn. checks	Total time	Total time
Hospital	`patsy`	3	0.45 ms	9.17 ms	6.38 ms
Blog	`post`	2	0.11 ms	1.03 ms	1.01 ms
	`comment`	3	0.29 ms	1.30 ms	1.01 ms

Table 2.2 gives the overall execution times for both the original system and the modified system. For the modified system, it also shows the number of stages for each transaction and the average time spent in dynamic checks for resolving potential stagepoints. The `comment` transaction in our experiments has one more stage than as described in Figure 2.11, because in all transactions, there is an initial stage performed to obtain the principals involved in the application.

By running the nodes on a single machine and using in-memory data storage, we maximize the fraction of the transaction run time occupied by dynamic checks. Nevertheless, this fraction remains small. While the effective low latency of communication between nodes reduces the overhead due to communication round-trips for staging precommits, we report the number of stages, from which this overhead can be calculated for arbitrary latency.

2.9 Related work

Various goals for atomic transactions, such as serializability [107] and ACID [65], have long been proposed and widely studied, and are still an active research

topic [114, 76, 135, 91, 13, 31]. While much of the recent interest has been focused on performance [50, 88, 148, 4, 156, 151], we focus on security.

Information leaks in commonly used transaction scheduling protocols have been known for at least two decades [135, 12]. Kang and Keefe [76] explore transaction processing in databases with multiple security levels. Their work focuses on a simpler setting with a global, trusted transaction manager. They assume each transaction has a single security level, and can only "read down" and "write up."

Smith et al. [135] show that strong atomicity, isolation, and consistency guarantees are not possible for all transactions in a generalized multilevel secure database. They propose weaker guarantees and give three different protocols that meet various weaker guarantees. Their Low-Ready-Wait 2PL protocol is similar to SC, and provides only what the authors call ACIS$^-$–correctness. Specifically, "aborted operations at a higher level may prevent all lower level operations from beginning" [135, p37]. Although our implementation is conservative and would not allow such a thing, the theory behind SC could allow a later stage with less trustworthy participants to hold up earlier, precommitted stages indefinitely.

Duggan and Wu [51] observe that aborts in high-security subtransactions can leak information to low-security parent transactions. Their model of a single, centralized multilevel secure database with strictly ordered security levels is more restrictive than our distributed model and security lattice. Our abort channels generalize their observation. They arrive at a different solution, building a theory of secure nested transactions.

Cohen, van der Meyden, and Zuck observe abort-based information leakage in the context of transactional memory [36]. They show how scheduling proto-

cols with "lazy invalidation conflict" and "lazily aggressive arbitration" [124] can schedule transactions securely. Their model assumes each transaction runs at a single security level, on a single processor. Relaxed monotonicity generalizes these constraints, allowing transactions to be distributed, and run at multiple security levels.

Atluri, Jajodia, and George [11] describe a number of known protocols requiring weaker guarantees or a single trusted coordinator. Our work instead focuses on securely serializing transactions in a fully decentralized setting. Our analysis is also the first in this vein to consider liveness: SC can guarantee deadlock freedom of transactions with relaxed monotonicity.

In this work, we build on a body of research that uses lattice-based information flow labels and language-based information flow methods [46, 48, 118]. Relatively little work has studied information flow in transactional systems. Our implementation is built on Fabric [91, 9], a distributed programming system that controls information flow over persistent objects. The only other information-flow-sensitive database implementation appears to be IFDB [122], which also does not account for abort channels.

2.10 Discussion

There is a fundamental trade-off between strong consistency guarantees and strong security properties in decentralized systems with heterogeneous participants. We investigate the secure scheduling of transactions, a ubiquitous building block of modern large-scale applications. Abort channels offer a stark example of an unexplored security flaw: existing transaction scheduling mechanisms can leak confiden-

tial information, or allow unauthorized influences of trusted data. While some sets of transactions are impossible to serialize securely, we demonstrate the viability of secure scheduling.

We present relaxed monotonicity, a simple condition under which secure scheduling is always possible. Our staged commit protocol can securely schedule any set of transactions with relaxed monotonicity, even in an open system. To demonstrate the practical applicability of this protocol, we adapted the Fabric compiler to check transactional programs for conditions that allow secure scheduling. These checks are effective: the compiler identifies an intrinsic security flaw in one program, and accepts other, secure transactions with minimal adaptations.

This work sheds light on the fundamentals of secure transactions, and builds toward a future that embraces heterogeneous components while preserving security. However, there is more work to be done to understand the pragmatic implications. We have identified separate necessary and sufficient conditions for secure scheduling, but there remains space between them to explore. Ultimately, abort channels are just one instance of the general problem of information leakage in distributed systems. Similar channels may exist in other distributed settings, and we expect it to be fruitful to explore other protocols through the lens of information flow analysis.

Acknowledgments

The authors would like to thank the anonymous reviewers for their suggestions. This work was supported by MURI grant FA9550-12-1-0400, by NSF grants 1513797, 1422544, 1601879, by gifts from Infosys and Google, and by the De-

partment of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program.

CHAPTER 3

CHARLOTTE

A Framework for Composable Authenticated Distributed Data Structures

Synopsis

We present *Charlotte*, a framework for composable, authenticated distributed data structures that embraces least ordering, fault tolerance, and heterogeneity. Charlotte data is stored in *blocks* that reference each other by hash. Together, all Charlotte blocks form a directed acyclic graph, the *blockweb*; all observers and applications use subgraphs of the blockweb for their own data structures. Unlike prior systems, Charlotte data structures are composable: applications and data structures can operate fully independently when possible, and share blocks when desired. To support this composability, we define a language-independent format for Charlotte blocks and a network API for Charlotte servers.

An authenticated distributed data structure guarantees that data is immutable and self-authenticating: data referenced will be unchanged when it is retrieved. Charlotte extends these guarantees by allowing applications to plug in their own mechanisms for ensuring availability and integrity of data structures. Unlike most traditional distributed systems, including distributed databases, blockchains, and distributed hash tables, Charlotte supports heterogeneous trust: different observers may have their own beliefs about who might fail, and how. Despite heterogeneity of trust, Charlotte presents each observer with a consistent, available view of data.

We demonstrate the flexibility of Charlotte by implementing a variety of integrity mechanisms, including consensus and proof of work. We study the power of

disentangling availability and integrity mechanisms by building a variety of applications. The results from these examples suggest that developers can use Charlotte to build flexible, fast, composable applications with strong guarantees.

3.1 Introduction

A variety of distributed systems obtain data integrity assurance by building distributed data structures in which data blocks are referenced using collision-resistant hashes [112], allowing easy verification that the correct data has been retrieved via a reference. We call these *Authenticated Distributed Data Structures* (ADDSs). A particularly interesting example of an ADDS is a blockchain, but there are other examples, such as distributed hash tables as in CFS [42], distributed version control systems like Git [144], and file distribution systems like BitTorrent [37]. However, an ADDS does not automatically possess all properties needed by blockchains and other applications. An ADDS might fail to ensure availability, because a reference to data does not guarantee it can be retrieved. It might even fail to ensure integrity, because an ADDS might be extended in inconsistent, contradictory ways—for example, multiple new blocks could claim to be the 7th in some blockchain.

Therefore, an ADDS commonly incorporates additional mechanisms to ensure availability and integrity in the presence of malicious adversaries. Some systems rely on gossip and incentive schemes to ensure availability, and consensus or proof-of-work schemes to ensure integrity. Blockchains like Bitcoin [106] and Ethereum [56] lose integrity if the adversary controls a majority of the hash power, while Chord loses availability if an adversary controls enough consecutive nodes [141].

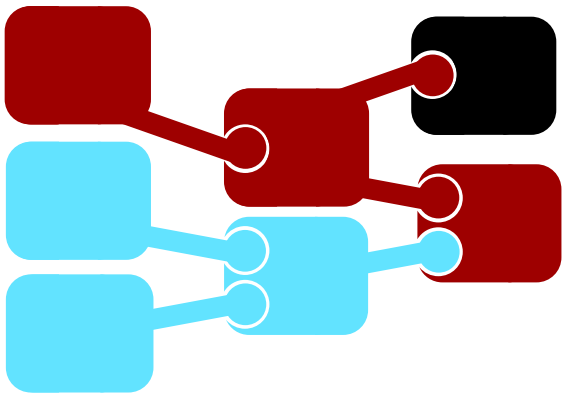

Figure 3.1: Blocks are represented as rectangles. References from one block to another are shown as circles. The pale blue blocks form a tree, whereas the darker red blocks form a chain. The rightmost red block references a blue block, so together the union of the red and blue blocks forms a larger tree. The black block also references a red block.

Importantly, all past ADDS systems lack *composability*: an application cannot use multiple ADDSs in a uniform way and obtain a composition of their guarantees. ADDSs from different systems cannot intersect (share blocks) or even reference each other. Lack of composability makes it difficult for applications to atomically commit information to multiple ADDSs. For instance, if blockchain ADDSs were composable, we could atomically commit a single block to two cryptocurrency blockchains, instead of requiring trusted clearinghouses.

A core reason for this lack of composability is that each system has its own set of failure assumptions. A user of Bitcoin or Ethereum, for example, must assume that at least half the hashpower is honest.[1] There is no mechanism for observers or applications to choose their own assumptions.

We address these limitations with *Charlotte*, a decentralized framework for composable ADDS with well-defined availability and integrity properties. Together, these ADDSs form the *blockweb*, an authenticated directed acyclic graph (DAG) [95] of all Charlotte data, which is divided into *blocks* that reference each

[1] There is some evidence that users need even stronger assumptions [53].

other by hash. Charlotte distills ADDSs down to their essentials, allowing it to serve as a common framework for building a wide variety of ADDSs in a composable manner, as illustrated by Fig. 3.1.

Within the blockweb, different applications can construct any acyclic data structure from blocks, including chains, trees, polytrees, multitrees, and skiplists. Charlotte embraces the Least Ordering principle (§ 1.1.1). Whereas blockchains enforce a total ordering on all data, the blockweb requires ordering only when one block references another. Unnecessary ordering is an enormous drain on performance; indeed, it arguably consumes almost all of traditional blockchains' resources. Charlotte applications can create an ordering on blocks, but blocks are by default only partially ordered.

In Charlotte, each server stores whichever blocks it wishes. Most servers will want blocks relevant to applications they're running, but some may provide storage or ordering as a service for sufficiently trusting clients.

Charlotte heterogeneous observers can set their own (application-specific) failure assumptions. The failure assumptions of an observer effectively filter the blockweb down to blocks forming an ADDS that remains available and consistent under all tolerable failures and adversarial attacks. An observer whose failure assumptions are correct can, given the assumptions of a different correct observer, calculate the subgraph of the blockweb they share.

A key novelty of Charlotte is its generality; it is not application-specific. Unlike other systems that build DAGs of blocks, Charlotte does not implement a cryptocurrency [110, 89, 115, 136, 137], require a universal "smart contract" language for all applications [66, 147, 73], have any distinguished "main chain" [109, 150],

or try to enforce the same integrity requirements across all ADDSs in the system [77, 93, 152, 43, 25].

Instead, Charlotte distills ADDSs down their essentials, allowing it to serve as a more general *ADDS framework*, in which each application can construct an ADDS based on its own trust assumptions and guarantees, yet all of these heterogeneous ADDSs are part of the same blockweb. Indeed, existing block-DAG systems can be recreated within Charlotte, gaining a degree of composability. We have implemented example applications to demonstrate that Charlotte is flexible enough to simultaneously support a variety of applications, including Git-like distributed version control, timestamping, and blockchains based variously on agreement, consensus, and proof-of-work. The shared framework even supports adding shared blocks on multiple chains.

Contributions

- Our mathematical model for ADDSs (§ 3.3) gives a general way to characterize ADDSs with diverse properties in terms of observers, a novel characterization of different failure tolerances for different participants, and a general way to compose ADDSs and their properties.
- Charlotte provides an extensible type system for blocks, and a standard API for communicating them (§ 3.4).
- Example applications show the benefits of using the Charlotte model (§ 3.5).
- We generalize blockchains in the Charlotte model, including a technique for separating availability and integrity duties onto separate services and a general model of linearizable transactions on distributed objects (§ 3.6).

- We have implemented a prototype of Charlotte along with proof-of-concept implementations of various applications that demonstrate its expressiveness and ability to compose ADDSs (§ 3.7).
- Performance measurements show that Charlotte's performance overheads are reasonable (§ 3.8).
- Our consensus-blockchain implementation uses Heterogeneous Consensus, and is evaluated in § 4.9.1.
- Analysis of real usage data shows that Charlotte's added concurrency offers a large speed advantage over traditional blockchain techniques (§ 3.6.5).

3.2 Overview

3.2.1 Blocks

In Charlotte, blocks are the smallest unit of data, so clients don't fetch "block headers," or other partial blocks [56]. Therefore, Charlotte applications ideally use small blocks. For instance, to build something like Ethereum in Charlotte, it would be sensible to create the Merkle tree [99] structure found within each Ethereum block out of many small Charlotte blocks. This makes it easier to divide up storage duties and to fetch and reference specific data.

3.2.2 Attestations

Some blocks are *attestations*: they prove that an ADDS satisfies properties beyond those inherent to a DAG of immutable blocks. For instance, if a server signs

an attestation stating that it will store and make available a specific block, it means the block will be available as long as that server functions correctly. Such an attestation functions as a kind of proof premised on the trustworthiness of the signing server. Attestations about the same blocks naturally compose: all properties of all attestations hold when all conditions are met.

All attestation types are *pluggable*: Charlotte servers can define their own subtypes, which prove nothing to observers who do not understand them. Charlotte is extremely flexible: application-defined attestation types can represent different consensus mechanisms (from Paxos to Nakamoto), different ADDS types, and different availability strategies. Although attestations can express a wide variety of properties about an ADDS, we divide them into two subtypes: *availability attestations* and *integrity attestations*.

3.2.3 Availability Attestations

Availability attestations prove that blocks will be available under certain conditions. One example of an availability attestation would be a signed statement from a server promising that a given block will be available as long as the signing server is functioning correctly. We call servers that issue availability attestations *Wilbur servers*.[2] Attestations may make more complex promises. For example, proofs of retrievability [23] might be used as availability attestations. Availability attestations are not limited to promises to store forever: they might specify any conditions, including time limits or other conditions under which the block is no longer needed. Availability attestations generalize features found in many existing distributed data systems:

[2]after the *Charlotte's Web* character whose objective is to stay alive.

- In BitTorrent, a seeder tells a tracker that it can provide certain files to leechers.
- Many databases inform clients that their transaction has been recorded by a specified set of replicas.
- In existing blockchains, clients wait for responses from many full nodes, to be sure their transaction is "available."

3.2.4 Integrity Attestations

An ADDS often requires some kind of permission to add a block to its state. For example, a blockchain typically requires that some set of servers ("miners") decide that a particular block uniquely occupies a given height in the chain. Integrity attestations determine which blocks belong in which ADDSs. For instance, servers maintaining a blockchain might issue an integrity attestation stating that a given block belongs on the chain at a specific height; the server promises not to issue any integrity attestation indicating that a different block belongs on the chain at that height. Timestamps are another integrity attestation type: they define an ADDS consisting of all blocks a specific server claims existed before a specific time. We call servers that issue integrity attestations *Fern servers*.[3]

Fern servers generalize ordering or consensus services. In blockchain terminology [106], they correspond to "miners," which select the blocks belonging on the chain.

[3]after the *Charlotte's Web* character who decides which piglets belong.

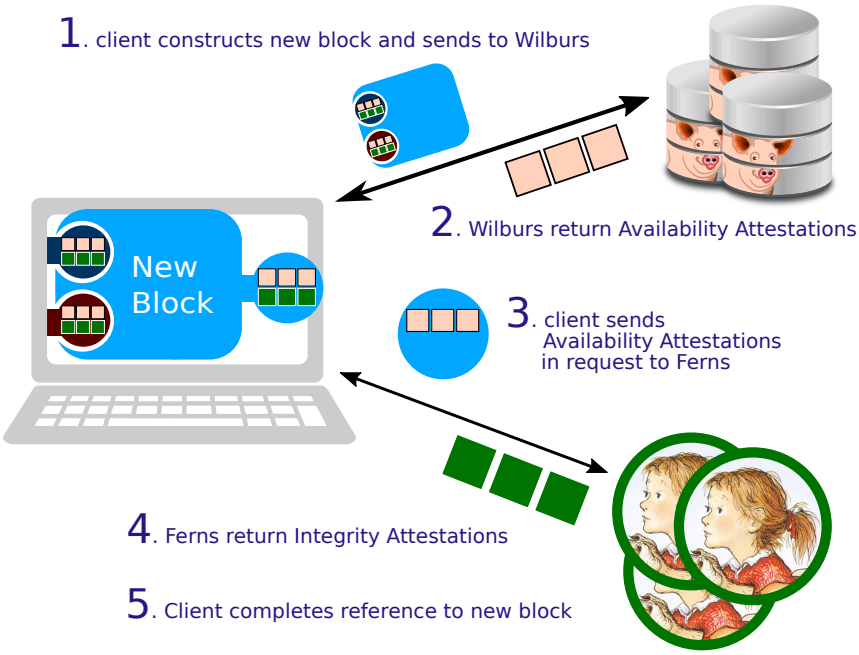

Figure 3.2: Life of a block. A client mints a new block, and wants to add it to an ADDS. The block, as drawn, includes two references to other blocks. The client acquires availability attestations from Wilbur servers, and integrity attestations from Fern servers. Then it can create a reference (drawn as a circle) to the block, so anyone observing the reference knows the block is in the ADDS.

3.2.5 Life of a Block

Fig. 3.2 illustrates one possible process for adding a new block to an ADDS. A client first mints a block, including data and references to other blocks. To ensure the block remains available, the client sends it to Wilbur servers, which store it and return availability attestations, demonstrating the availability of the block.

The client then submits a reference to the block to a collection of *Fern* servers, which maintain the integrity of the ADDS. Since Fern servers may not want to permanently add a block to their ADDS if that block is going to become unavailable, the client may also send availability attestations. Fern servers return *integrity*

attestations, that, in effect, demonstrate the integrity of the statement "this block is in this ADDS."

The client includes all of these attestations in references to the block, so that whenever an observer sees a reference to the block, they know how available it is, and what ADDSs it belongs to. Over time, more attestations may be issued, so a block can become more available or join more ADDSs, with greater integrity.

Charlotte is flexible: applications can optimize this process by co-locating services, forwarding attestations directly between servers, etc.

3.2.6 Observers

We characterize an *observer* in a distributed system as an entity with a set of assumptions concerning the possible ways that the system can fail. Note that failure types include both Crash and Byzantine [82]. Given a set of assumptions about who can fail and how, and the desired integrity properties of each ADDS, each observer may choose to ignore any portions of the blockweb that lack adequate attestations. What remains is the observer's *view* of the ADDS: the set of blocks it believes are available and part of the state of the ADDS.

Each observer's view of an ADDS is guaranteed to remain available and to uphold any integrity properties the observer has chosen so long as the observer's failure assumptions hold. Further, portions of the blockweb that feature attestations satisfying two observers are guaranteed to remain in both observers' views, once both have observed all the relevant blocks. Of course, in practice, servers take time to download relevant blocks, and in an asynchronous system there is no bound on the time this may take.

3.2.7 Example Applications

Blockchains

Charlotte can easily represent blockchains—not only linear chains, but also more intricate sharded or DAG-based structures [95]. Existing blockchain systems already effectively provide integrity and availability attestations, phrased as proofs of work, proofs of stake, etc. Charlotte makes these proofs more explicit, without limiting the attestation types an application can use. As a result, multiple chains can share a block, if attestations required for each all refer to the same block. By providing a framework in which applications can interact, but without prescribing a rigid data structure, Charlotte allows far more concurrency than monolithic chains like Ethereum that totally order all blocks into a single chain [56]. This flexibility is a natural realization of the database community's decades-old ideal of imposing a "least ordering" [18].

Distributed Version Control

Charlotte is also a natural framework for applications like Git [144]. Each Git *commit* is a block referencing zero or more parent commits. A commit with multiple parents is a *merge*, and a commit with no parents is a *root*. Each Git server stores and makes some commit blocks available, and can communicate this fact with availability attestations. A Git server can also maintain *branches*, which associate a branch name (a string) with a chain of commits. When a server announces that it is making a new commit the head of a branch, it issues an integrity attestation stating that the commit is part of the branch.

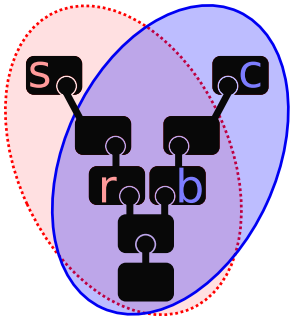

Figure 3.3: A blockchain ADDS with branches of length < 3. All of the blocks are present in the blockweb. The red dotted oval and the blue solid oval represent two possible *alternative* states of the ADDS.

Public-Key Infrastructure

Public Key Infrastructure (PKI) systems are almost always ADDSs. Key endorsements are essentially integrity attestations, defining ADDSs such as the certificate trees used to secure HTTPS [70] and the web of trust used to secure PGP [28]. Keys and certificates can be retrieved by hash from dedicated storage servers such as PGP's keyservers [67, 69, 127], corresponding to Wilbur servers. PKIs such as ClaimChain [80] already attest to and rely upon data structure properties, e.g., total ordering in chains.

3.3 Modeling ADDSs Formally

Different, possibly overlapping, portions of the blockweb represent ADDSs of interest to individual applications. We now explore Charlotte's unique ability to allow different ADDSs to interoperate.

As a running example, consider a simple ADDS R representing a single, write-once slot managed by one server. It can either be empty, or occupied by one unchanging block.

3.3.1 States

A *state* is a set of blocks, and an *ADDS* is a set of possible states. For instance, the Bitcoin blockchain is an ADDS. Every block (other than the origin) in every state features a proof-of-work. A Bitcoin state can have an arbitrarily long main chain, and shorter branches. The Bitcoin ADDS consists of all such possible states.

In our single-slot example, each state of R is either empty, or features exactly two blocks: the block occupying the slot, along with an integrity attestation signed by the server, referencing that block. We call an integrity attestation in such a state i_x, where x is the other block in the state.

3.3.2 Observers and Adversaries

Observers represent principals who use the system. An observer receives blocks from servers and in so doing learns about the current and future states of ADDSs in the system. Observers may correspond (but are not limited) to servers, clients, or even people. Formally, an observer is an agent that *observes* an ordered sequence of blocks from the blockweb. On an asynchronous network, different observers may see different blocks in different orders.

Observers define their own failure assumptions, such as who they believe might crash or lie. These assumptions, combined with evidence, in the form of blocks they have observed so far, induce an observer's *belief*: what they think is true about the blockweb now and what is (still) possible in the future.

The failure-tolerance properties of any distributed system are relative to assumptions about possible failures, including actions taken by adversaries. Char-

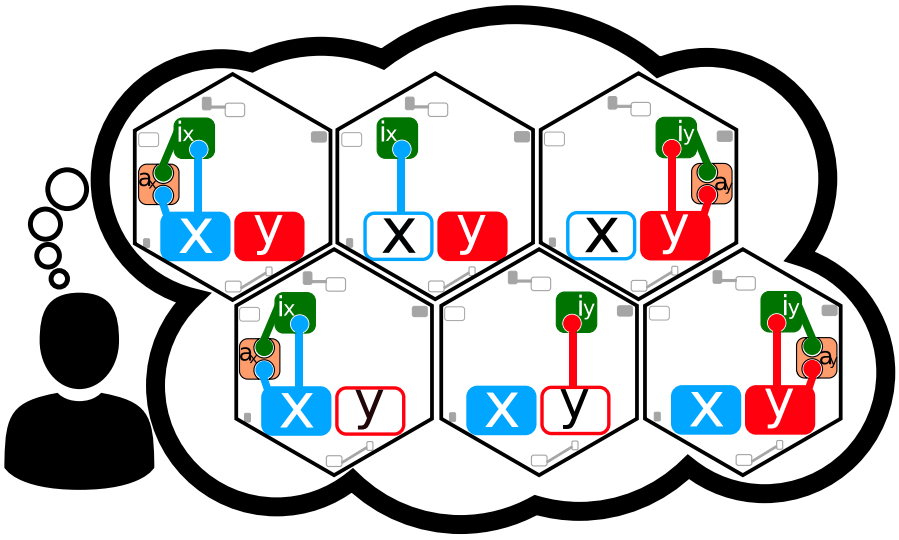

Figure 3.4: An *observer* holds a *belief*, which is a set of *universes*. Here we've drawn some universes as hexagons. Each universe U shown contains blocks in $exist(U)$, with blocks in $avail(U)$ filled in.

---

lotte makes these assumptions explicit for each observer. An observer who makes incorrect assumptions may not observe the properties they expect of some ADDSs. For instance, if more servers are Byzantine than the observer thought possible, data they believed would remain available might not. Alternatively, data structures might lose integrity, such as when two different blocks both appear to occupy the same height on a chain.

We characterize a belief α as a set of possible *universes*. This set bounds the believed powers of the adversary: the observer assumes this set includes all possible universes that might occur under the influence of the adversary. Fig. 3.4 illustrates an observer holding a belief, and some of the universes in that belief.

Each observer has an *initial belief*: the belief it holds before it observes any blocks. For example, an observer who trusts one Fern server to maintain the single-slot ADDS R does not have any universes in its initial belief in which that server has issued two integrity attestations for different blocks. This belief encodes

the observer's assumption that the server's failure isn't tolerable. The observer in Fig. 3.4 has such a belief: no universe features two integrity attestations for R (shown as green squares labeled i_x or i_y).

In a traditional failure-tolerant system, an observer usually assumes that no more than f participants will fail in some specific way (e.g., crash failures or Byzantine failures). We model such an observer's initial belief as the set of all universes in which no more than f participants exhibit failure behaviors (in the form of blocks issued).

3.3.3 Formalizing Universes

We propose a general model for universes that places few limits on the details or assumptions universes can encode. Our model of a universe U has the following components, which suffice for all examples in the paper.

1. A set of blocks that can *exist*, written $exist(U)$. These are the blocks that either have already been observed or ever *can* be observed by any observer.
2. A *strict partial order* $\underline{\underline{\sqsubset_U}}$ on $exist(U)$. Every observer is assumed to observe blocks in an order consistent with the universal partial order $\underline{\underline{\sqsubset_U}}$.
3. The set of blocks that *are available*, written $avail(U)$. These are the blocks that can be retrieved from some server. Any available block must also exist: $avail(U) \subseteq exist(U)$.

The set $exist(U)$ constrains the blocks any observer will observe. It does not model time: an observer's initial belief contains universes representing all possible futures, with all blocks that are possible in each.

Since we are modeling asynchronous systems, the model does not explicitly include the time when blocks are observed, but the ordering $\underline{\overline{U}}$ constrains the times at which different observers can observe blocks, implicitly capturing a temporal ordering on blocks. This ordering is useful for blockchains like Bitcoin, where observers traditionally do not believe in any universe U unless there is a *main chain* in which each block b is ordered (by $\underline{\overline{U}}$) before any equal-height block with which b does not share an ancestor fewer than security parameter k (usually 6) blocks away. Further, the *main chain* must forever outpace any other branch. In Fig. 3.3, this belief (with $k = 3$) implies that if a Bitcoin observer believes in a universe U in which both blocks s and c exist, they must be ordered by $\underline{\overline{U}}$. If Bitcoin's security assumptions are correct, any two observers must see s and c in the same order.

We make the simplifying assumption in each of our example applications that the only availability of interest is permanent: we want to characterize whether blocks will forever be available. Hence, the set $avail(U)$ increases over time. We leave more nuanced availability policies to future work.

3.3.4 Updating Beliefs

As an observer observes blocks being created by Charlotte programs, it updates its beliefs by whittling down the set of universes it considers possible. For instance, if an observer with belief α observes a block b, clearly b *can exist*, so the observer

refines its belief. It creates a new belief α', filtering out universes in which b is impossible:

$$\alpha' = \left\{ U \;\middle|\; b \in \mathit{exist}(U) \;\wedge\; U \in \alpha \right\}$$

If the observer in Fig. 3.4 were to observe i_x, it would update its belief, retaining only universes U with $i_x \in U$. Of the universes shown, only the leftmost three would remain.

An observer also refines its belief by observation order: If an observer with belief α observes blocks B in total order $<_B$, then its new belief is:

$$\mathit{Possible}(\alpha, B, <_B) \triangleq \left\{ U \;\middle|\; \begin{array}{l} \forall b' \underline{\boxed{U}}\; b.\; b' \in B \;\wedge\; b' <_B b \\ \wedge\, B \subseteq \mathit{exist}(U) \\ \wedge\, U \in \alpha \end{array} \right\}$$

An observer making no assumptions believes in all possible universes. It can only eliminate universes inconsistent with its observations: those in which blocks it has observed are impossible, or the order in which it has observed the blocks is impossible. However, most interesting observers have other assumptions. For example, the observer in Fig. 3.4 trusts that only one integrity attestation for ADDS R will be issued, so if it observes i_x and removes all universes U without $i_x \in \mathit{exist}(U)$, then no universes with i_y will remain.

As another example, when a Git observer observes a valid integrity attestation for a block b, it can eliminate all universes with valid integrity attestations for blocks that are not descendants or ancestors of b.

3.3.5 Observer Calculations

An observer with belief α knows a set of blocks B are *available* if they're made available in all possible universes:

$$\forall U \in \alpha.\ B \subseteq \mathit{avail}(U)$$

For example, the observer in Fig. 3.4 trusts availability attestations a_x and a_y (the orange squares): it does not believe in any universe where such attestations reference an unavailable block.

Likewise, an observer with belief α knows a state S of an ADDS D is *incontrovertible* if no *conflicting* state S' can exist in any possible universe. Two states conflict if they cannot be merged to form a valid state: observing one precludes ever observing the other:

$$\forall U \in \alpha, S' \in D.\ (S \cup S' \in D)\ \vee\ (S' \not\subseteq \mathit{exist}(U))$$

For example, the observer in Fig. 3.4 trusts that only one integrity attestation for ADDS R will be issued. It does not believe in any universes with both i_x and i_y (shown as green squares). Therefore, if it observes i_x, it knows the state $\{i_x, x\}$ is incontrovertible: no conflicting state (such as $\{i_y, y\}$) exists in any universe in its belief.

The state of ADDS D that an observer with belief α sees as available and incontrovertible is therefore:

$$\mathit{View}(\alpha, D) \triangleq$$
$$\bigcup \left\{ S \;\middle|\; \begin{array}{l} \quad \forall U \in \alpha, S' \in D.\ (S' \subseteq S) \vee (S' \not\subseteq \mathit{exist}(U)) \\ \wedge\ \forall U \in \alpha.\ S \subseteq \mathit{avail}(U) \\ \wedge\ S \in D \end{array} \right\}$$

We call this the observer's *view* of the ADDS: Charlotte's natural notion of the "current state." So long as an observer's assumptions are correct, new observations can only cause its view to grow. For example, if the observer in Fig. 3.4 observes both a_x and i_x, then it believes the state $\{i_x, x\} \in R$ is available and incontrovertible. Its view of the single-slot ADDS R features x occupying the slot, and so long as its assumptions are correct, this will never change.

As another example, suppose a blockchain uses a simple agreement algorithm: a quorum of servers must attest to a block being at a specific height. States consist of a chain of blocks, each with integrity attestations from a quorum. An observer's view will not include any blocks lacking sufficient attestations. The observer assumes that no two blocks with the same height both get a quorum of attestations, so the chain it has viewed must be a prefix of the chain in any future view.

One observer can calculate what another observer's view of an ADDS would be, if they see the same observations. When two observers communicate, they can share blocks they've observed. Because new observations can only cause a view to grow, this allows one observer to know (at least part of) another observer's view when they communicate. This what we mean when we say views in Charlotte are *consistent:* two observers can know what the other views in the same data structure, and so the state of a data structure can be, in a sense, global.

3.3.6 Composability

Recall that a *state* is a set of blocks, and an ADDS is a set of states (§ 3.3.1). ADDSs in Charlotte have two natural notions of composition: *union* ($\uplus$) and *intersection* ($\pitchfork$).

Union

Intuitively, the union of two ADDSs D and D' is all the data in either ADDS. As states are sets of blocks (§3.3.1), their union is simply the traditional union of sets. Thus, the union ADDS is composed of unions of states:

$$D \uplus D' \triangleq \left\{ \; S \cup S' \;\middle|\; S \in D \;\wedge\; S' \in D' \; \right\}$$

As a result, given an observer's failure assumptions, its view of the union of two ADDS is simply the union of its views of the ADDSs:

Theorem 5.

$$\forall \alpha, D. \mathit{View}(\alpha, D \uplus D') = \mathit{View}(\alpha, D) \cup \mathit{View}(\alpha, D')$$

Proof. Follows from the definitions of *View* and $\uplus$. □

For example, a Git branch (§3.2.7) is a ADDS maintained by one server. A Git repository is the *union* of many branches with the same root, on the same server. Each branch ADDS has properties, such as linearity, not necessarily shared by the repository as a whole. However, the properties of all the ADDSs in a *union* can be combined to create properties that hold of the whole. For example, one server makes available all the blocks in all the branches of a repository. That means that

the repository remains available so long as the server is correct. See § 3.3.5 for more details.

Intersection

Intuitively, the intersection of two ADDSs D and D' is all the data that is in both D and D'. As states are sets of blocks (§ 3.3.1), their intersection is simply the traditional intersection of states. Thus, the intersection of ADDSs is composed of the intersections of states:

$$D \pitchfork D' \triangleq \left\{ S \cap S' \;\middle|\; S \in D \;\wedge\; S' \in D' \right\}$$

As a result, given an observer's failure assumptions, its view of the intersection of two ADDS is simply the intersection of its views of the ADDSs:

Theorem 6.

$$\forall \alpha, D. \mathit{View}(\alpha, D \pitchfork D') = \mathit{View}(\alpha, D) \cap \mathit{View}(\alpha, D')$$

Proof. Follows from the definitions of *View* and $\pitchfork$. □

For example, consider two blockchains, each serving as a ledger for a different crypto-currency. The blocks that are part of both chains represent transactions atomically committed to both ledgers. These are the natural place to put *cross-chain transactions*: trades involving both crypto-currencies. Thus, the intersection of the two blockchains is the sequence of cross-chain transactions.

The intersection ADDS shares the properties of all intersected ADDSs. In our blockchain example, the cross-chain blocks remain totally ordered by the blockweb so long as either component blockchain remains totally ordered by the blockweb

(a traditional integrity property of blockchains). Furthermore, cross-chain blocks remain available so long as the blocks of either component blockchain remain available. See § 3.3.5 for more details.

3.3.7 Availability Attestation Semantics

Observers use availability attestations to determine which blocks they consider sufficiently available to be in ADDSs they care about (§ 3.2.3). Formally, availability attestations guarantee some blocks are available in some universes. To describe the guarantees offered by an availability attestation x, we give it an interpretation $[\![x]\!]$ that is a *belief*: that is, a set of universes in which x's guarantees are inviolate (§ 3.3.2).

For instance, consider the availability attestation subtype $\tau_{AliceProvides}$. Attestations of this type are blocks of the form $aliceProvides(b)$ (where b is another block). Intuitively, each value states that `Alice` (a Wilbur server) promises to make the specified block b available forever. Thus, all universes U in which $aliceProvides(b)$ exists also have b available:

$$[\![aliceProvides(b)]\!] \triangleq \left\{ U \;\middle|\; aliceProvides(b) \in exist(U) \;\Rightarrow\; b \in avail(U) \right\}$$

Defining attestations this way makes it easy to define observers' beliefs based on which attestations, attestation types, or even participants they trust. For instance, if an observer trusts all attestations with type τ, we define that observer's belief:

$$\alpha = \bigcap_{x:\tau} [\![x]\!]$$

This provides a straightforward definition for what it means to *believe in* a type; it means trusting all attestations of that type.

$$[\![\tau]\!] \triangleq \bigcap_{x:\tau} [\![x]\!]$$

We can also define beliefs that trust only combinations of attestations. For example, if an observer believes a block will be available only if it has observed appropriate attestations of both type τ *and* type σ, we define that belief α as $\alpha = [\![\tau]\!] \cap [\![\sigma]\!]$.

Likewise, a more trusting observer who believes a block is available if it has observed appropriate attestations of type τ *or* type σ would believe $\alpha = [\![\tau]\!] \cup [\![\sigma]\!]$. In this way, we can even build up quorums of attestations or attestation types (e.g., $([\![\tau_1]\!] \cap [\![\tau_2]\!]) \cup ([\![\tau_2]\!] \cap [\![\tau_3]\!]) \cup ([\![\tau_1]\!] \cap [\![\tau_3]\!])$).

There are some restrictions on the semantics of an availability attestation type. Attestations must be *monotonic*: adding more attestations never proves weaker statements:

$$\begin{aligned} \forall U, V, W \in [\![\tau]\!].\ \ & \mathit{exist}(U) \cup \mathit{exist}(V) \subseteq \mathit{exist}(W) \ \Rightarrow \\ & \mathit{avail}(U) \cup \mathit{avail}(V) \subseteq \mathit{avail}(W) \end{aligned}$$

3.3.8 Integrity Attestation Semantics

Integrity attestations (§ 3.2.4) are issued by Fern servers (§ 3.4.2), and represent proofs guaranteeing the *non-existence* of other integrity attestations, under certain circumstances. While this definition may seem counter-intuitive, it generalizes the notion of *conflict* or *exclusivity* in ADDSs. For example, in our single-slot ADDS R, all the integrity attestations found in any state of R are mutually exclusive.

Since each (non-empty) state of R contains an integrity attestation, the existence of one attestation disproves all conflicting states, which puts the attestation, and the block it references, in the view of any observer with an appropriate belief.

Formally, an integrity attestation guarantee some other blocks *will not exist* in some universes.

Thus, we represent every attestation's meaning as a set of universes, essentially a *belief* (§ 3.3.2) in that attestation. To describe integrity attestations' guarantees, we have a static semantics where attestations are identified with *beliefs*, sets of universes in which the integrity attestation's guarantees are inviolate (§ 3.3.2).

For example, consider $\tau_{BobCommits}$, a subtype of integrity attestation with values that are blocks of the form $bobCommits(b)$, which intuitively indicates that `Bob` (a Fern server) promises never to commit to any block other than b. These integrity attestations are much like the ones used in our single-slot ADDS R.

Thus, all universes U in which $bobCommits(b) \in exist(U)$ don't feature $bobCommits(c)$ for any $c \neq b$:

$$[\![bobCommits(b)]\!] \triangleq \left\{ U \;\middle|\; \forall c \neq b \;.\; bobCommits(c) \notin exist(U) \right\}$$

Defining attestations this way makes it easy to define observers' beliefs based on which attestations, attestation types, or even participants they trust. For instance, if an observer trusts all attestations with type τ, we define that observer's belief:

$$\alpha = \bigcap_{x:\tau} [\![x]\!]$$

This provides a straightforward definition for what it means to *believe in* a type; it means trusting all attestations of that type.

$$[\![\tau]\!] \triangleq \bigcap_{x:\tau} [\![x]\!]$$

We can also define beliefs that trust only combinations of attestations. For instance, an observer who believes b is committed only after receiving an attestation of type τ and an attestation of type σ would believe $\alpha = [\![\tau]\!] \cup [\![\sigma]\!]$. Likewise, a more trusting observer who believes b is committed after receiving an attestation of either type τ or σ would believe $\alpha = [\![\tau]\!] \cap [\![\sigma]\!]$.

It is also possible to combine integrity and availability attestations to define a belief. An observer who trusts attestations of x to commit blocks, and attestations of y to ensure their availability would believe: $\gamma = [\![x]\!] \cap [\![y]\!]$. In this way, we can even define quorums of trusted attestations or attestation types.

The definition of $Possible([\![\tau]\!], B, <_B)$ (from § 3.3.2) guarantees integrity attestation semantics are *monotonic*: adding more attestations never proves weaker statements:

$$C \subseteq B \Rightarrow Possible(\tau, B, <_B) \subseteq Possible(\tau, C, <_B)$$

3.3.9 Implementation Limitations of Attestations

Since programmers can define their own subtypes of integrity or availability attestations, nothing prevents them from encoding availability guarantees in an integrity attestation, or violating the availability attestation monotonicity requirement (§ 3.3.7). Programmers who violate the system assumptions naturally lose guarantees.

In our implementation, the only operational distinction between an availability attestation and an integrity attestation is in the `Reference` object. When one block references another, it can also reference relevant integrity and availability

```
message AnyWithReference {
 google.protobuf.Any any;
 Reference typeBlock;}
message Hash {
 oneof hashalgorithm_oneof
   { AnyWithReference any;
     bytes sha3; }}//technically unnecessary
message Reference {
 Hash hash;
 repeated Hash availabilityAttestations;
 repeated Reference integrityAttestations;}
message Block {
 oneof blocktype_oneof
  { AnyWithReference any;
    string protobuf; }}
```

Figure 3.5: Core Types of Charlotte: this (slightly simplified) proto3 code describes how blocks, references to blocks, and generic data are safely marshaled and unmarshaled in Charlotte.

attestations. However, whereas an included reference to an integrity attestation is itself a `Reference` object, an included reference to an availability attestation carries only a `Hash`. This is because an integrity attestation might need an availability attestations to describe where to obtain the integrity attestation. However, the same is not true of an availability attestation: it is pointless to send availability attestation b just to describe where to fetch availability attestation a, since it is just as easy to send availability attestation a in the first place.

3.4 Charlotte API

Charlotte is a set of protocols by which clients, Fern servers, and Wilbur servers interact. Different servers can run different implementations of these protocols. Our implementation of Charlotte (§ 3.7) uses gRPC [64], a popular language-independent network service specification language, based on Protocol

Buffers [113]. Hence, we use Protocol Buffer (protobuf) syntax to describe the Charlotte protocols.

Fig. 3.5 presents the core types used by Charlotte protocols, using Protocol Buffer syntax.[4] Charlotte is built around these core types:

- **`Block`**: can contain any protobuf [113] data type, or the block itself can be a protobuf type definition. `Attestation` is a subtype of `Block`. It can contain any protobuf [113] data type, and the block itself can be a protobuf type definition.
- **`Hash`**: represents the hash of a block.
- **`Reference`**: is used by one block to reference another; it contains the `Hash` of the referenced block, along with zero or more references to attestations (§ 3.2.2).
- **`AnyWithReference`**: Anyone can add their own subtypes of `Block`, `Hash`, or `Attestation`, which any server can safely marshal and unmarshal. It contains a reference to the block where the type description can be found (as proto3 [113] source code), and marshaled data.[5]

In practice, we provide some useful example subtypes of `Hash` (e.g., `sha3`) and `Block` (e.g., `Attestation`).

In our API, all Charlotte servers must implement the `SendBlocks` RPC (Fig. 3.6), which takes in a stream of blocks and can return a stream of responses

[4] For simplicity, our specifications omit the indices of the various fields. The actual source code is also slightly more complicated for extensibility (`https://github.com/isheff/charlotte-public`).

[5] The proto3 `Any` type itself features a URL string meant to reference the type definition, but Charlotte uses a block reference because it is self-verifying.

```
message SendBlocksResponse {
 string errorMessage;}
service CharlotteNode {
 rpc SendBlocks(stream Block)
     returns (stream SendBlocksResponse) {}}
```

Figure 3.6: All Charlotte servers implement the CharlotteNode service.

```
message AvailabilityPolicy {
 oneof availabilitypolicytype_oneof {
  AnyWithReference any; }
}
message RequestAttestationResponse {
 string errorMessage;
 Reference reference;
}
service Wilbur {
 rpc RequestAvailabilityAttestation(
      AvailabilityPolicy)
     returns (RequestAttestationResponse){}
}
```

Figure 3.7: Wilbur Service Specification.

that may contain error messages. We define subtypes of attestation for *Availability* and *Integrity*, and show how to construct and observer from quorums of types they trust (§ 3.3.7 and § 3.3.8).

3.4.1 Wilbur

Wilbur servers host blocks, providing *availability*.

In blockchain terminology [106], Wilbur servers correspond to "full nodes," which store blocks on the chain. In more traditional data store terminology, Wilbur servers are key–value stores for immutable data. The Charlotte framework is intended to be used for building both kinds of systems.

```
message IntegrityPolicy {
 oneof integritypolicytype_oneof
  { AnyWithReference any; }
}
service Fern {
 rpc RequestIntegrityAttestation(
      IntegrityPolicy)
     returns (RequestAttestationResponse){}
}
```

Figure 3.8: Fern Service Specification.

In our API, Wilbur servers are Charlotte servers that include the `RequestAvailabilityAttestation` RPC (Fig. 3.7), which accepts a description of the desired attestation, and returns either an error message, or a reference to a relevant availability attestation.

3.4.2 Fern

Fern servers issue *integrity attestations*, which define the set of blocks in a given ADDS. Among other things, integrity attestations can be proofs-of-work, or records demonstrating some kind of consensus has been reached. One simple type of integrity attestation, found in our prototype, is a signed pledge not to attest to any other block as belonging in a specific slot in an ADDS. Fern servers generalize ordering or consensus services. In blockchain terminology [106], Fern servers correspond to "miners," which select the blocks belonging on the chain.

In our API, Fern servers are Charlotte servers that include the `RequestIntegrityAttestation` RPC (Fig. 3.8), which accepts a description of the desired attestation, and returns either an error message or a reference to a relevant integrity attestation.

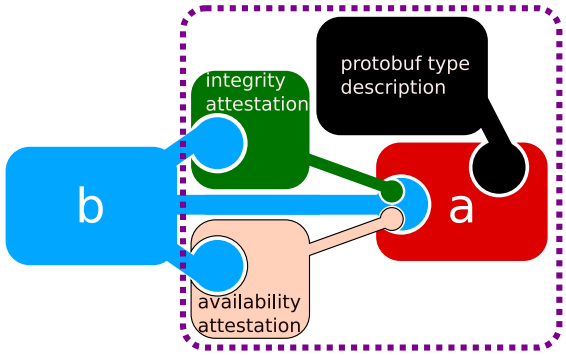

Figure 3.9: Block a references block b, and that reference carries attestations. Block a also references a type description block, for unmarshaling data in a. In general, when sending block a to a server or client, the sender should be sure the recipient has received *all* the blocks in the dashed purple rectangle, so the recipient can fully understand block a and the properties of its references.

3.4.3 Practices for Additional Properties

In order to understand a reference object within a block (how available the referenced block is, and data structures it's in), an observer reads attestations referenced within the reference object.[6] For example, without the content of the availability attestations, it's not clear where to look to retrieve the referenced block. As a rule of thumb, before one server sends a block to another, it should ensure the recipient has any attestations or type blocks referenced within that block. This ensures the recipient can, in a sense, fully understand the blocks they receive. In Fig. 3.9, for instance, when sending block a, the sender should be sure the recipient has received everything in the dashed rectangle. Our example applications follow this practice. It is possible, however, that for some applications, servers may be certain the recipient doesn't care about some attestations or type blocks, and therefore might leave those out.

[6] We considered making references contain full copies of attestations, but this made blocks large, and since many blocks may reference the same block (and attestations), blocks were full of redundant information.

When servers follow this practice, it's useful for availability attestations to attest to groups of blocks likely to be requested together. In Fig. 3.9, for instance, and availability attestation that attests to everything in the dashed rectangle would be more useful than just attesting to block a. Our example applications' availability attestations are generally designed this way.

Availability failures can cause available states of ADDSs to become disconnected subgraphs (if the blocks that connect them are forgotten). To build an ADDS that will always remain connected, availability attestations that attest to a block should also attest to the availability attestations referenced within that block. Furthermore, whenever a block x references a block y, and block y references block z, if y isn't at least as available as z, then x should reference z as well. (Here, "isn't at least as available" means that the availability attestations in references to z guarantee z will be available in some universe where the availability attestations in references to y do not guarantee y will be available.)

3.5 Use Cases

In addition to the examples mentioned earlier (§ 3.2.7), Charlotte is well-suited to a wide variety of applications.

3.5.1 Verifiable Storage

Our Wilbur specification provides a common framework for verifiable storage. Because ADDS references include hashes, it is always possible to check that data

```
message WilburQueryInput {
 oneof wilburquery_oneof {
  Reference reference = 1;
  Block fillInTheBlank = 2; }}
message WilburQueryResponse {
 string errorMessage = 1;
 repeated Block block = 2; }

service WilburQuery {
  rpc WilburQuery(WilburQueryInput)
      returns (WilburQueryResponse) {}}
```

Figure 3.10: WilburQuery Specification.

retrieved was the data referenced. Furthermore, availability attestations (§ 3.3.7) are a natural framework for proofs of retrievability [23].

Queries

In addition to `SendBlocks` and `RequestAvailabilityAttestation`, Wilbur servers may offer other interfaces. Application designers may wish to implement query systems for retrieving relevant blocks. We created one such example interface, the `WilburQuery` RPC (Fig. 3.10). Given a `Hash` as input, `WilburQuery` returns the block with that hash. If the server does not know of such a block, our example implementation waits until one arrives.

`WilburQuery` also provides a kind of fill-in-the-blank match: If sent a block with some fields missing, `WilburQuery` returns the all stored blocks that match the input block in the provided fields. For example, we might query for all blocks with a field marking them as a member of a certain ADDS.

3.5.2 Timestamping

Timestamps are a subtype of integrity attestation. We implemented a Signed Timestamp type, wherein the signer promises that they have seen specific hashes before a specific time. Our timestamping Fern servers can use *batching*: they wait for a specific (configurable) number of new requests to arrive before issuing a Timestamp block referencing all of them. In fact, since hash-based references represent a *happens-before* relationship [81], timestamps are transitive: if timestamp a references timestamp b, and b references c, then a effectively timestamps c as well.

We recommend that batch Timestamp blocks themselves should be submitted to other Timestamping Fern servers. This allows the tangled web of Timestamp blocks to very quickly stamp any block with exponentially many timestamps, making them very high-integrity.

3.5.3 Conflict-Free Replicated Data Types

Charlotte, and ADDSs in general, work well with CRDTs, especially Operation-Based Commutative Replicated Data Types (CmRDTs) [126]. CmRDTs are replicated objects maintained by a group of servers. Whenever a new operation originates at any server, all known operations on that object on that server are said to *happen before* it. Then the operation asynchronously propagates to all other servers. Thus, the set of operations known to any particular server are only partially ordered. The *state* of a CmRDT object is a deterministic function of a set of known operations (and their partial order). For example, a CmRDT implementa-

tion of an insert-only Set might feature the `insert` operation, and its state would be the set of all arguments to known `insert` operations.

In Charlotte, CmRDT operations can naturally be expressed as blocks, with *happens-before* relationships expressed as references. Since references are by hash, it is impossible for an adversary to insert a cycle into the graph of operations. The states of a CmRDT can be formally expressed as all possible sets of operations with all possible partial orderings.

Aside from whatever credentials one needs to authorize an operation, CmRDTs do not need integrity attestations. Observers need only consider the graph of known, valid operation blocks with known ancestry. They may, of course, choose to filter out blocks they consider insufficiently available. Availability attestations are still useful.

The blockweb as a whole is a CmRDT: Its state is the DAG of all blocks, and every block is an operation adding itself to the state. Other than the blockweb itself, however, we have not implemented any interesting CRDTs yet.

3.5.4 Composition

Charlotte ADDSs are easy to compose (§ 3.3.6). At the most basic level, blocks in one ADDS can reference blocks in another. For instance, a Timestamp server might maintain a chain of timestamp blocks, which reference any other blocks people want timestamped (Git commits, payments, documents, etc.). A Git-style repository might reference earlier commits in another repository (either because one is a fork of the other, or one has merged in code from another) without having to copy all of the data onto both servers. This would resemble Git's *submodule* sys-

tem [32]. A blockchain could reference a Git commit as a smart contract, instead of hosting a separate copy of the code [56]. A single block of data, stored on some highly available servers, could be referenced from a variety of torrent-style file-sharing applications, git-style repositories, and blockchains, without unnecessary duplication.

At a high level, composability allows us to build high-integrity ADDSs out of low-integrity ones (§3.3.6). For instance, the blocks that appear in the intersection of two chains form a chain that can only fork if *both* component chains fork. Users may want to put especially important blocks on many different chains, the way they want many different witnesses for important legal transactions.

Likewise, we can build low-integrity ADDSs out of higher integrity ones (§3.3.6). If a set of blockchains each manage independent tokens, and sometimes share blocks (for atomic trades of tokens), then together all the chains form a DAG. If *any* chain in the DAG is corrupted, then the supply of that token may not be conserved: the DAG as a whole is lower integrity than any one chain. This makes it possible to talk about the "integrity of the marketplace" as distinct from the integrity of any one token.

3.5.5 Entanglement

Some attestations, such as timestamps §3.5.2, and proofs of work §3.6.2, implicitly lend integrity to everything in a block's ancestry. When many ADDSs reference each other's blocks, these recursive attestations can make some forms of fraud very difficult. For example, if many applications regularly reference past timestamps, and many applications request timestamps from a variety of servers, it quickly

becomes difficult to falsely claim a block *did not* happen before a given time, when doing so would involve hiding evidence embedded in many different applications.

3.6 Blockchains as ADDSs

Charlotte is an ideal framework for building new blockchains and related applications (§ 3.2.7). In the simplest sense, a blockchain is any path through the blockweb. However, most existing blockchain applications are considerably more complicated.

Like all ADDSs, a blockchain needs *integrity* and *availability*. Here, integrity means that an observer's view (§ 3.3.2) always features a *main chain*, in which no two blocks ever have the same *height*. Availability means that once an observer observes a main chain block at a height, that block remains available for download indefinitely.

3.6.1 Separating Availability and Integrity

With few exceptions [97], existing blockchain systems require that all integrity servers (e.g., miners, and consensus nodes) store all the blockchain data. This is fundamentally inefficient. For example, a traditional byzantine consensus system tolerating f failures needs $> 3f$ participants, while a storage system tolerating f failures needs only $> f$ participants. If blockchain systems separated storage and consensus duties, they would be able to store about 3 times as much as they do, with the same failure assumptions.

Charlotte makes it easy to separate availability from integrity. Wilbur servers store blocks, and provide availability attestations (§ 3.4.1). References to those blocks carry those attestations, proving the block referenced is available. Fern servers need only issue integrity attestations for each block on the chain, rather than storing it themselves.

For example, if one were to build something like Ethereum in Charlotte, what Ethereum calls *block headers* would themselves be integrity attestations, and the Merkle root in each would instead be a reference (or collection of references) to blocks stored on Wilbur servers. This makes it natural to search and retrieve block headers and portions of state, without splitting apart blocks, or downloading the whole chain.

3.6.2 Integrity Mechanisms

Different blockchains have used a variety of mechanisms to maintain the integrity of the chain [106, 96, 26]. To demonstrate the flexibility of Charlotte, we have implemented a few example mechanisms in small-scale experiments.

Nakamoto (Proof-or-Work)

We can represent a Bitcoin/ Ethereum style blockchain as an ADDS D whose states are trees of proof-of-work blocks. An observer with security parameter k (say, 6) believes only in universes with a main chain that grows faster than any side chains differing by k or more blocks. More precisely, if a universe U includes a state S of such a blockchain D featuring a fork of k or more blocks, one side of

the fork must be the *main* chain. All main chain blocks k or higher above the root of the fork must be observed before $\left(\overline{\underline{U}}\right)$ all other blocks in S of equal height.

Agreement

Some blockchain applications only require agreement: they lose liveness if two *potentially* valid blocks are proposed for the same height [115, 89, 136]. For instance, if a chain represents a single bank account, and potentially valid blocks represent transactions signed by the account holder, then honest account holders should never sign two transactions unordered by the blockweb.

Agreement servers are simple to implement. When a server attests to a block, it promises never to attest to any conflicting block. For a given server, an agreement attestation type τ does not feature any universes where two conflicting blocks both have an attestation from the server. Observers can construct quorums of trusted servers, as in § 3.3.8. A block appears in an observer's view when the observer has observed enough attestations: committing a conflicting block would require too many parties to break their promises.

3.6.3 Blocks on Multiple Chains

In general, nothing prevents a single block from being part of multiple chains. It simply requires the integrity attestations for each chain. For example, if one blockchain represents records of events that have happened to a specific vehicle (crashes, repairs, ...), and another represents repairs a specific vendor has performed, it makes sense to append the record of a specific repair to both chains. The record (a block) could reference the previous blocks on each chain, and the

next blocks on each would in turn reference it. Each chain's integrity mechanism would have to attest to the block, and references to the block could carry both sets of attestations to let readers know it is in both ADDSs.

Atomicity

Sometimes, such block appends need *atomicity*. For example, suppose one blockchain represents the cryptocurrency RedCoin, and another represents the cryptocurrency BlueCoin. Alice wants to give Bob one RedCoin in exchange for one BlueCoin. This represents two transactions: one on each chain. Crucially, either *both* happen, or *neither* do. Otherwise, it's possible that Alice will give Bob a RedCoin, and get nothing in return. We want to commit both transactions together, *atomically*.

Meet

To atomically commit one block to multiple ADDSs, we require a single integrity attestation which represents a commitment to all of them. We call the type of this integrity attestation the *meet* ($\sqcap$) of the types of the integrity attestations for the ADDSs involved. If an attestation of type τ_r commits a block to RedCoin, and an attestation of type τ_b commits a block to BlueCoin, then an attestation of type $\tau_r \sqcap \tau_b$ commits a block to both. In a sense, $\tau_r \sqcap \tau_b$ is a subtype of τ_r or τ_b, since an attestation of the meet type can be used wherever an attestation of either supertype can. In our types-as-observers semantics (§ 3.3.7), we define meet as $\sqcap \triangleq \cap$. The assumptions made by the meet type encapsulate all the assumptions made by its component types.

Not all pairs of integrity attestation types have a meet. Later, in chapter 4, we define a consensus algorithm with a natural meet operation (§ 4.9.2).

3.6.4 Linearizable Transactions on Objects

It can be useful to model state as a collection of stateful *objects*, each of which has some availability and integrity constraints [114]. We can model objects as a chain of blocks, defined by availability and integrity attestations upholding these constraints. For instance, if an object must be consistent and available so long as 3 of a specific 4 servers are correct, each block should have "store forever" availability attestations from 2 servers, and integrity attestations from 3 stating that they'll never attest to any other block in that slot.

Each block represents a state change for each of the objects represented by chains of which the block is a part. In other words, the blocks are *atomic* (or *ACID*) *transactions* in the database sense [65]. A collection transactions is guaranteed to have a consistent, serial order so long as the chains maintained for each of the objects they touch are consistent. For a given observer, the transactions involving objects which that observer assumes to be linearizable have a serial order so long as that observer's assumptions are correct. Furthermore, two correct observers can never see two transactions oppositely ordered.

This gives programmers a natural model for atomic transactions across object-chains with different integrity and availability mechanisms, which would be useful for applications from banking to supply chain tracking. Transactions can involve any set of objects, so long as their integrity mechanisms have a meet operation for atomic commitment (§ 3.6.3).

Banking

We can imagine bank accounts as a linearizable objects, with state changes being deposits and withdrawals to and from other bank accounts, signed by appropriate parties. We can model this in Charlotte. Each bank maintains some integrity mechanism (Fern servers) to ensure accounts' state changes are totally ordered, which prevents double-spending. Likewise, each bank maintains some Availability mechanism (Wilbur servers), ensuring transactions relevant to their customers' accounts aren't forgotten. Each transaction is thus a block shared by two chains, and must be committed atomically onto both chains.

When considering how "trustworthy" the money in an account is, what matters is the integrity of the ADDS featuring the full ancestry of all transactions in the account. To ensure the trustworthiness of their accounts, banks may issue their own integrity attestations for all transactions in the causal past of transactions involving that bank. This requires checking that ancestry for any inconsistencies with anything to which the bank has already attested. This ensures any observers trusting the bank's attestations have consistent view (§3.3.2), but cannot guarantee that observers trusting different banks have the same view.

An "attest to the complete history" approach is analogous to auditing the full finances of everyone with whom you do business for every transaction. In reality, much of the time, banks effectively trust each other's attestations. This allows much faster transaction times with weaker guarantees.

Table 3.1: Theoretical advantages of Charlotte-style parallelization in the Bitcoin payment network

		Unaltered	**2 Accounts**
linearized	longest chain	6,953,512	24,129,215
	time	**3.72 years**	12.91 years
parallelized	longest chain	110,787	244,163
	time	21.63 days	**47.68 days**

Supply Chain Tracking

Much like bank accounts, we can imagine each good in a supply chain as a linearizable object. Transactions may involve decreasing / destroying some goods to increase / create others. For example, a transaction might feature destroying 10 kg from a case of grapes to add 9 kg to a vat of juice, and 1 kg to a bin of compost. As with banking, each good is only as "trustworthy" as the ADDS featuring its complete ancestry, and audits / attesting to past transactions can increase this trustworthiness.

3.6.5 Application to Payment Graphs

The Charlotte framework makes it easy to imagine parallelized blockchain-based payments, with each account as a stateful object, represented by a chain (§ 3.6.4). As the Bitcoin payment network is a popular example of blockchain-based finance, we consider the theoretical advantages offered by parallelization in a Charlotte-style approach.

Bitcoin does not keep track of money in terms of accounts. Instead, each transaction divides all its money into a number of outputs, called Unspent Transaction Outputs, or UTXOs, each of which specify the conditions under which they can

be spent (e.g., a signature matching this public key). Each transaction specifies a set of input UTXOs as well, from which it gets the money, and it provides for each a proof that it is authorized to spend the money (e.g., a digital signature). Each UTXO is completely drained when it is spent, and cannot be reused. Thus, Bitcoin transactions form a graph, with transactions as vertices and UTXOs as directed edges [106].

In our Charlotte banking model, each bank account is a chain, so a transfer between two accounts is simply a block on two chains (§ 3.6.4). Therefore, if two sets of financial transactions don't interact, they can operate entirely in parallel. The speed of the system is limited by the speed of its slowest chain. If appending a transaction to its chains takes constant time, the speed limit is simply the length of the longest chain.

Blocks 1 through 200,000 of Bitcoin contain 6,953,512 transactions. The longest chain through this graph has length 110,787, so in principle, Charlotte needs time for only 110,787 rounds of consensus to accommodate the entire payment graph. Although Bitcoin batches several transactions per block, it required 200,000 rounds of consensus to do the same, taking a total of 3.72 years. Thus, even with a similarly slow consensus mechanism, a parallelized Charlotte approach, even with no batching, would require only 21.63 days. Of course, Charlotte bank accounts can specify Fern servers with whatever consensus mechanism they like. This could be a much faster system, such as PBFT [30].

In Bitcoin, it improves anonymity and performance to combine many small transfers of money into big ones, with many inputs and many outputs. In the real financial system of the USA, however, all monetary transfers are from one account to another. They are all exactly two-chain transactions. We can simulate this lim-

itation by refactoring each transaction as a DAG of transactions with logarithmic depth (Appendix A.1).

With this construction, a Charlotte banking system might use more than one transaction per Bitcoin transaction. The longest chain through this new transaction graph has length 244,163; so, in principle, Charlotte can process the entire graph in only this many rounds of consensus. Thus, even with a consensus mechanism as slow as that of Bitcoin, Charlotte would still require only 47.68 days, a speedup of 28.

3.7 Implementation

Our full Charlotte spec, with all example types and APIs, is 298 lines of gRPC (mainly protobuf) [64]. We implemented proof-of-concept servers in 3833 lines of Java [63] (excluding comments and import statements), with a further 1133 lines of unit tests. We also wrote 1149 additional lines of Java setting up various experiments. This code inclues our Charlotte experiments with Heterogeneous Consensus, detailed in § 4.9.1. Our code is availble on Github at `https://github.com/isheff/charlotte-public`.

3.7.1 Wilbur servers

By default, our example Wilbur servers store all blocks received in memory forever. They are not meant to be optimal, but they are usable for proof-of-concept applications. The only type of availability attestation we have implemented is one in which the Wilbur servers promise to store the block indefinitely. This attesta-

```
message PublicKey {
  message EllipticCurveP256 {
    bytes byteString;}
  oneof keyalgorithm_oneof {
    AnyWithReference any;
    EllipticCurveP256 ellipticCurveP256;}}
message CryptoId {
  oneof idtype_oneof {
    AnyWithReference any;
    PublicKey publicKey;
    Hash hash;}}
message Signature {
  message SHA256WithECDSA {
    bytes byteString;}
  CryptoId cryptoId;
  oneof signaturealgorithm_oneof {
    AnyWithReference any;
    SHA256WithECDSA sha256WithEcdsa;}}
```

Figure 3.11: Signature Specification. We include `Any` types for extensibility, as well as default built-in types, like `Sha256WithECDSA`. Note that the `message` keyword defines a type in the local scope.

tion proves that the block is available as long as the Wilbur server is functioning correctly.

Our Wilbur servers can be configured with a list of known peers, to whom they will relay any blocks they receive and any attestations they create. This is easy to override: servers can be made to relay blocks to any collection of peers.

We also implemented the WilburQuery service of § 3.5.1. Our Wilbur servers can do fill-in-the-blank pattern matching on all implemented block types. The Wilbur Query service imposes no overhead on other services.

```
message SignedGitSimCommit {
  message GitSimCommit {
    message GitSimParents {
      message GitSimParent {
        Reference parentCommit;
        bytes diff;}
      repeated GitSimParent parent;}
    string comment;
    Hash hash;
    oneof commit_oneof {
      bytes initialCommit;
      GitSimParents parents;}}
  GitSimCommit commit;
  Signature signature;}

message Block {
  oneof blocktype_oneof {
    AnyWithReference any;
    string protobuf;
    SignedGitSimCommit signedGitSimCommit;}}

message IntegrityAttestation {
  message GitSimBranch {
    google.protobuf.Timestamp timestamp;
    string branchName;
    Reference commit;}
  message SignedGitSimBranch {
    GitSimBranch gitSimBranch;
    Signature signature;}
  oneof integrityattestationtype_oneof {
    AnyWithReference any;
    SignedGitSimBranch signedGitSimBranch;}}
```

Figure 3.12: Git Simulation integrity attestation Specification. We include `Any` types for extensibility, and provide types like `SignedGitSimBranch` as options. Note that the `message` keyword defines a type in the local scope, and that the `Signature` type is defined in the full Charlotte spec (`https://github.com/isheff/charlotte-public`).

3.7.2 Version Control

We implemented a simulation of Git [144]. Our servers are not fully-functional version control software, as they do not implement file-diffs and associated checks, which are irrelevant for the purpose of demonstrating the Charlotte framework.

The types for our version control ADDS are described in Fig. 3.12. We created a block subtype, `SignedGitCommit`, representing a specific state of the files tracked. Each block features a signature, comment, hash of the state. It can be an *initial* commit, in which case it has no parents, but does include bytes representing the full contents of the files being tracked. Alternatively, it can have some number of parent commits, each with a reference and a file diff.

A Version Control Fern server tracks the current commit it associates with each *branch* (strings). They issue integrity attestations that declare which commits they've put on which branches. A correct Fern server should never issue two such attestations for the same branch, unless the commits they reference are ordered by the blockweb. In other words, each new commit on a branch should follow from the earlier commit on that branch; it cannot be an arbitrary jump to some other files. Our example servers enforce this invariant (`https://github.com/isheff/charlotte-public`).

Fern servers can have other reasons to reject a request to put a commit on a branch. Perhaps they accept only commits signed by certain keys. When a client issues a request, they can include attestation references. A Fern server can demand that clients prove a commit is, for instance, stored on certain Wilbur servers before it agrees to put it on a branch. The Wilbur servers need not even be aware of the Git data types.

```
message IntegrityAttestation {
  message TimestampedReferences {
    google.protobuf.Timestamp timestamp;
    repeated Reference block;}
  message SignedTimestampedReferences {
    TimestampedReferences timestampedReferences;
    Signature signature;}
  oneof integrityattestationtype_oneof {
    AnyWithReference any;
    SignedTimestampedReferences sigTimeRefs;}}
```

Figure 3.13: Timestamping integrity attestation Specification. We include `Any` types for extensibility, and provide `SignedTimestampedReferences` as an option. Note that the `message` keyword defines a type in the local scope, and that the `Signature` type is defined in the full Charlotte spec (`https://github.com/isheff/charlotte-public`).

Our version control implementation can use the same Wilbur servers as any other application. In fact, separating out the storage duties of Wilbur from the branch-maintaining duties of Fern allows our Charlotte-Git system to divide up storage duties of large repositories, much like git-lfs [60].

3.7.3 Timestamping

Timestamps are a subtype of integrity attestation. Each Timestamp includes a collection of references to earlier blocks, the current clock time [72], and a cryptographic signature.

Our Timestamping Fern servers timestamp any references requested, using the native OS clock. By default, they issue a timestamp immediately for any request, and do not need to actually receive the blocks referenced. Because references contain hashes, the request itself guarantees the block's existence before that time.

Our Timestamping Fern servers also implement *batching*. Every 100 (configurable at startup) timestamps, the Fern server issues a new timestamp, referencing the blocks it has timestamped since the last batch. Each server then submits its batch timestamp to other Fern servers (configurable at startup) for timestamping. Since timestamps are transitive (if a timestamps b, and b references c, then a also timestamps c), blocks are very quickly timestamped by large numbers of Fern servers. This allows applications to quickly gather very strong timestamp integrity.

3.7.4 Blockchains

In principle, any path through the blockweb is a blockchain (§ 3.6). We implemented Fern servers using three very different integrity mechanisms (§ 3.6.2). We used some of these servers to demonstrate the advantages of separating integrity and availability mechanisms (§ 3.6.1), and blockchain composition: we put blocks on multiple chains (§ 3.6.3).

Agreement

Our Agreement Fern servers keep track of each a blockchain as a *root* block, and a set of *slots*. Each slot has a number representing distance from the root of the chain.

Our Agreement Fern servers use the `SignedChainSlot` subtype of integrity attestation (Fig. 3.14). It features a cryptographic signature, and references to a chain's *root*, a slot number, and the block in that slot. This serves as a format for both requests and attestations. Each request is simply an `IntegrityAttestation`

```
message IntegrityAttestation {
  message ChainSlot {
    Reference block;
    Reference root;
    uint64 slot;
    Reference parent;}
  message SignedChainSlot {
    ChainSlot chainSlot;
    Signature signature;}
  oneof integrityattestationtype_oneof {
    AnyWithReference any;
    SignedChainSlot signedChainSlot;}}
message IntegrityPolicy {
 oneof integritypolicytype_oneof
  { AnyWithReference any;
    IntegrityAttestation fillInTheBlank;}}
```

Figure 3.14: Agreement integrity attestation Specification. We include `Any` types for extensibility, and provide `SignedChainSlot` as an option. Note that the `message` keyword defines a type in the local scope, and that the `Signature` type is defined in the full Charlotte spec (`https://github.com/isheff/charlotte-public`).

with some fields (like the cryptographic signature) missing. While it is possible to encode this in the `IntegrityPolicy`'s `any` field, we provide the `fillInTheBlank` option as a convenience.

The Agreement Fern servers are configured with parameters describing which requests they can accept, in terms of requirements on the reference to the proposed block and its parent. Once a correct Agreement Fern server has attested that a block is in a slot, it will *never* attest that a different block is in that slot. For instance, to configure a blockchain using quorums of 3 Agreement Fern to approve each block, we require that each request's `parent` Reference include 3 appropriate integrity attestations.

Our Agreement Fern servers make it easy to separate integrity and Availability duties (§ 3.6.1). To ensure that a block is available before committing it to the

```
message IntegrityAttestation {
  message NakamotoIntegrityInfo {
    Reference block;
    Reference parent;}
  message NakamotoIntegrity {
    NakamotoIntegrityInfo info;
    uint64 nonce;}
  oneof integrityattestationtype_oneof {
    AnyWithReference any;
    NakamotoIntegrity nakamotoIntegrity;}}
```

Figure 3.15: Nakamoto integrity attestation Specification. We include `Any` types for extensibility, and provide `NakamotoIntegrity` as an option. Note that the `message` keyword defines a type in the local scope (`https://github.com/isheff/charlotte-public`).

chain, we require a `block` Reference to include specific availability attestations from Wilbur servers.

Nakamoto

Nakamoto, or *Proof of Work* Consensus is the integrity mechanism securing Bitcoin [106]. We model it formally in § 3.6.2. In Bitcoin, *miners* create proofs of work, which are stored by *full nodes*. With the Simplified Payment Verification (SPV)protocol, *clients* submit a transaction, and retrieve the *block headers* (proofs of work and Merkle roots) of each block in the chain from full nodes [106]. Each client can use these to verify that its transaction is in the chain (has integrity).

We implement miners as Fern servers, which produce integrity attestations bearing proofs of work, taking the place of block headers. Wilbur servers take the place of full nodes, and store blocks, including integrity attestations. For simplicity, our implementation assumes one transaction per block, so clients generate blocks,

and request attestations. When a client receives an integrity attestation (Fig. 3.15), it can retrieve the full chain from Wilbur servers.

With SPV, Clients traditionally try to collect block headers until they see their transactions buried "sufficiently deep" in the chain. For simplicity, our Fern servers delay responding to the client at all until the client's block has reached a specified (configurable) depth. Regardless, clients can collect integrity attestations from Wilbur servers until they're satisfied.

Our implementation of Nakamoto consensus offers a more precise availability guarantee than Bitcoin does. Nakamoto Fern servers demand availability attestations with any blocks submitted, ensuring that before a block is added to the chain, it meets a (configurable) availability requirement.

3.8 Evaluation

To evaluate the performance of Charlotte, we ran instances of each example application. Except as specified, experiments were run on a local cluster using virtual machines with Intel E5-2690 2.9 GHz CPUs, configured as follows:

- Clients: 4 physical cores, 16 GB RAM
- Wilbur servers: 1 physical core, 8 GB RAM
- Fern servers: 1 physical core, 4 GB RAM

To emulate wide area communication, we introduced 100 milliseconds artificial communication latency between VMs.

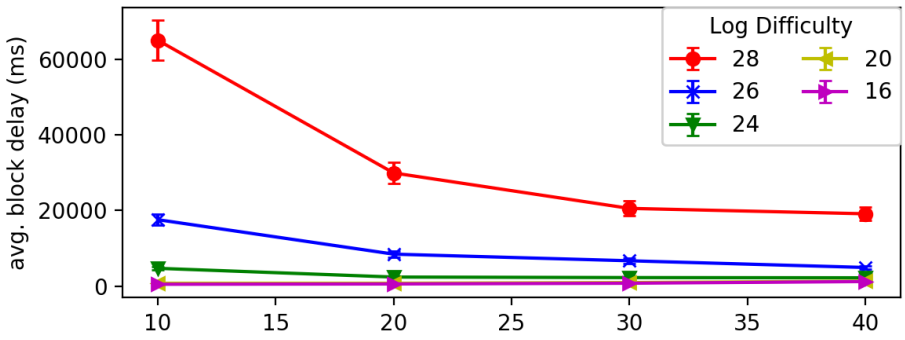

Figure 3.16: Mean block delay of Nakamoto on Charlotte, with bars showing standard error. Difficulty is represented in log_2 of the number of hashes expected to mine a new block.

3.8.1 Blockchains

Since blockchains are an obvious application of Charlotte, we evaluated the performance, scalability, and compositionality of various blockchain implementations.

Nakamoto

To compare performance of our Nakamoto implementation to Bitcoin's, we used multiple ($n = 10, 20, 30, 40$) Charlotte nodes and measured the mean delay (across 100 consecutive blocks) until a client received an integrity attestation for a block with fixed security parameter $k = 1$. All clients and servers had one physical core, and 4 GB RAM. Fig. 3.16 shows the results of our tests with various difficulty values (expected number of hashes to mine a block).

When difficulty is low, the delay for an integrity attestation is dominated by the communication overhead (200 ms). When, more realistically, the difficulty is high, delay is dominated by the cost of mining. Fig. 3.16 shows that latency increases with difficulty and decreases with the inverse of the number of Charlotte servers

(total computational power). Charlotte indeed scales suitably for blockchain implementations.

In fact, Bitcoin has about 2×10^{11} times the hash power [44], and 10^{14} times the difficulty as we had in our experiment, and it achieves an average block latency of 10 min. With compute power scaled appropriately, our implementation would achieve comparable performance: about 5 minutes per block.

Agreement

To evaluate the bandwidth advantages of separating integrity and availability services, we built Agreement Chains (§ 3.7.4) tolerating 1–5 Byzantine failures, both with and without Wilbur servers. To tolerate f Byzantine failures, a chain needs $3f + 1$ Fern servers, and, if it relies on Wilbur servers for availability, $f + 1$ Wilbur servers. We tested the latency and bandwidth of our chains, with some experiments using 10 byte blocks, and some using 1 MB blocks. In each experiment, a single client appends 1000 blocks to a chain, with the first 500 excluded from measurements to avoid warm-up effects. Each experiment ran three times.

In the simple case, without Wilbur servers, all Fern servers receive all blocks. This resembles the traditional blockchain strategy [106]. The theoretical minimum latency is 2 round trips from the client to the Fern servers, or 200 ms.

We also built chains that separate the Fern servers' integrity duties from Wilbur servers' availability duties. In these chains, Fern servers would not attest to any reference unless it included $f + 1$ different Wilbur servers' availability attestations.

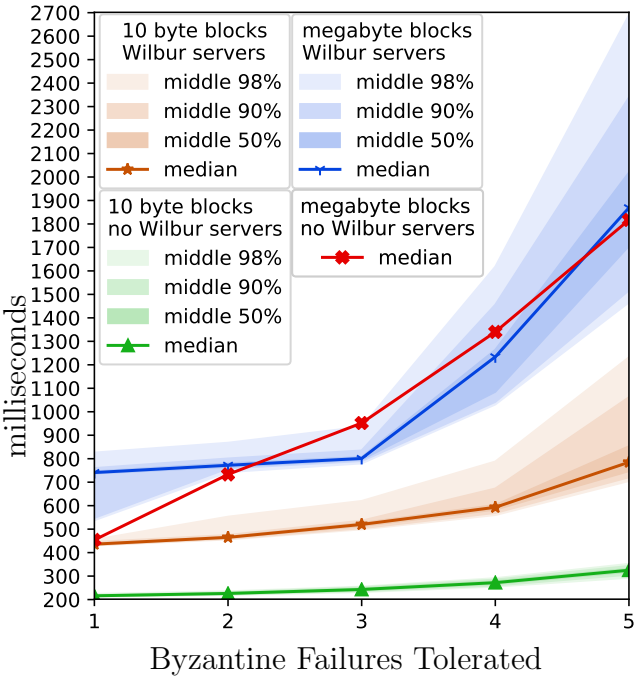

Figure 3.17: Time to commit blocks in Agreement chains with various numbers of servers. The shaded zones cover the middle percentile of blocks, so the top of the lightest zone represents the 99th (slowest) percentile, and the bottom represents the 1st (fastest) percentile. The distribution for the megabyte-block, no-wilbur-server experiment is in Fig. 3.18.

Latency Fig. 3.17 and Fig. 3.18 show the median latency to commit a block for each of our Agreement chain experiments. Theoretical minimum latency is 4 message sends (round trip from the client to the Wilbur servers, and then from the client to the Fern servers), or 400 ms. For chains with small blocks, latency remains close to the 200 ms and 400 ms minimums. For chains with 1 megabyte blocks, experimental setup has significant slowdowns, likely due to bandwidth limitations.

Bandwidth Separating availability and integrity concerns (§3.6.1) has clear benefits in terms of bandwidth. Because it sends large blocks to just $f + 1$ Wilbur servers instead of $3f + 1$ Fern servers, our client uses much less bandwidth in the large-block experiments with Wilbur servers than without them (Fig. 3.19).

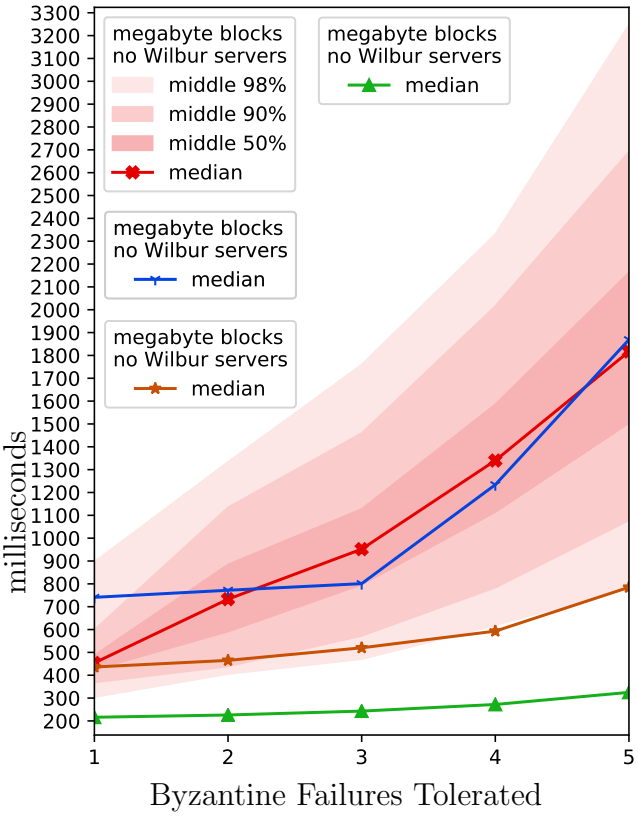

Figure 3.18: Time to commit blocks in Agreement chains with various numbers of servers. The distribution for the megabyte-block, no-wilbur-server experiment is shown.

In theory, committing a block with Wilbur servers requires bandwidth for $f + 1$ blocks, and without Wilbur servers requires $3f + 1$ blocks. The overhead inherent in the additional communication with Wilbur servers and the attestations issued is small compared to the savings.

3.8.2 Timestamping

To evaluate performance, compositionality, and entanglement (§ 3.5.5) with a non-blockchain application, we ran experiments with varying numbers of Timestamping Fern servers (§ 3.7.3). All client and server VMs had 4 GB RAM. For each ex-

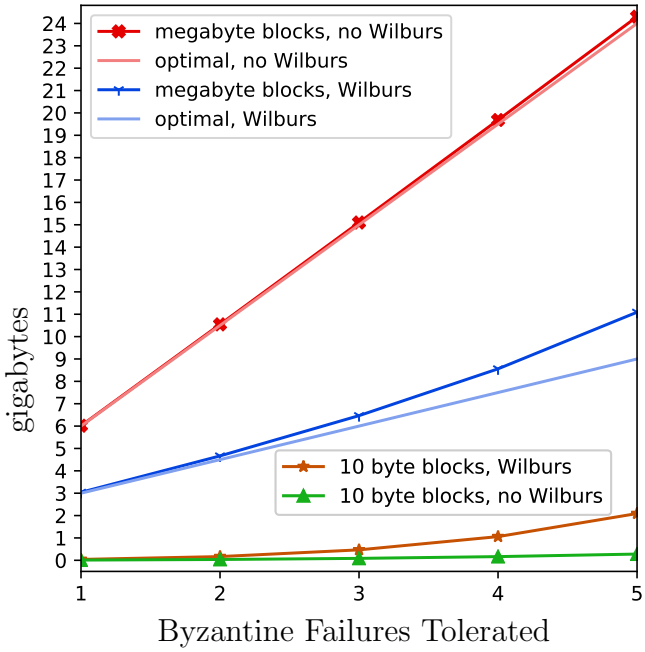

Figure 3.19: Total bandwidth used by a client appending 1500 blocks to Agreement-based chains.

---

periment, a single client requested timestamps for a total of 100,000 blocks. For each block, it requested a timestamp from one server, rotating through the Fern servers.

For each 100 timestamps a Fern server issued, it would create a new block referencing those 100 timestamps, and request that all other Fern servers timestamp this block. Since timestamps are transitive (if c is a timestamp referencing b, and b references a, then c also timestamps a), every block was soon timestamped by all Fern servers.

To explore Charlotte's compositionality, we also composed our (1- or 2-failure-tolerant) Agreement chains with our Timestamping Fern servers. We saw no statistically significant change in chain performance: the overhead of Timestamping was

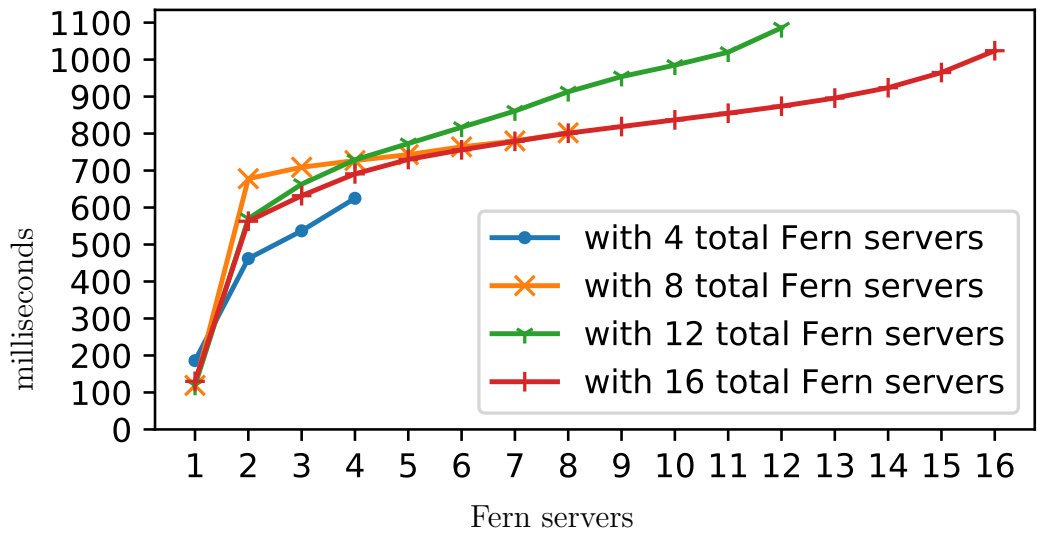

Figure 3.20: Mean time for a block to be timestamped by x Fern servers, in experiments featuring 4, 8, 12, and 16 total Fern servers.

---

unmeasurably small. Each block was timestamped quickly by directly requested Timestamping servers, but entanglement (§ 3.8.2) was limited by the chain rate.

We also calculated the time it took blocks to accrue different Fern servers' timestamps. As Fig. 3.20 shows, the Fern servers quickly timestamp each request. Blocks get 1 timestamp very close to the 100 ms network latency minimum. There is a delay between 1 and 2 timestamps because it takes a little while for the Fern servers to collect 100 timestamps and to create their own block. After that, blocks accrue timestamps very quickly, since each Fern Server requests timestamps from all other Fern servers. These experiments suggest that entanglement (§ 3.5.5) can be a fast, efficient, and compositional way to lend integrity to large ADDSs.

Not all blocks took exactly the same amount of time to accrue the same number of timestamps. Fig. 3.21 shows the distribution of times for blocks in the experiment with 16 Fern Servers. The scale is the same as in Fig. 3.20. In general, each

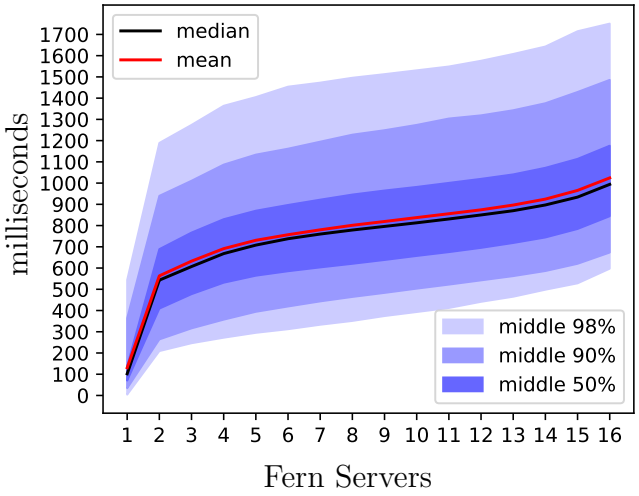

Figure 3.21: Time for a block to be timestamped by x Fern Servers, in an experiment featuring a total of 16 Fern Servers. Shaded zones cover the middle percentile of blocks, so the top of the lightest zone represents the 99th (slowest) percentile, and the bottom represents the 1st (fastest) percentile. Also shown are mean and median block times (very similar).

---

data point (time for blocks to accrue x timestamps in an experiment with n Fern Servers) was approximately Poisson distributed.

3.9 Related Work

3.9.1 Address by Hash

Many other distributed systems reference content by hash, forming ADDSs. Most reference schemes, however, only work within a specific application. For instance, git uses hashes to reference and request commits stored on a server [144]. Git-lfs can track and request large files on separate servers with hash-based identifiers [60]. Similarly, PKI systems (§ 3.2.7) reference keys and certificates by hash, and maintain groups of availability servers [70, 67, 80]. Distributed Hash Tables, such as

CFS [42] ultimately maintain Availability servers, and ensure integrity by referencing data via Hash.

HTML pages can reference resources using the `integrity` field [149] to specify a hash, and the `src` field to specify a server, like an availability attestation without formal guarantees. Likewise, BitTorrent's Torrent files [37] and Magnet URIs [38] reference a file by hashes of various kinds, and can specify "acceptable sources" from which to download the file. Charlotte's references aim to be extensible in terms of the hash algorithms used, and generic over all types of data. Uniquely, Charlotte bundles references to data with references to attestations, which can offer precise formal guarantees.

In concurrent work, Protocol Labs' IPLD [74] is a multi-protocol format for addressing arbitrary content by hash. Like Charlotte's `AnyWithReference` (§ 3.4), Multiformats [104] offers an extensible format for self-describing data including protobufs [113]. Both IPLD and Multiformats are developed closely with IPFS [16], a peer-to-peer file distribution system. IPLD references do not include attestation references the way Charlotte references do, but future work could fruitfully combine these technologies with Charlotte's reference and block encoding formats.

3.9.2 BlockDAGs

Other projects have explored DAGs of blocks in a blockchain context. Many, such as Iota [110], Nano (also known as RaiBlocks) [89], Avalanche [115], Spectre [136], Phantom, and Ghostdag [137] are tailored to cryptocurrency. Each defines its own currency, and they do not compose.

Some projects, such as æternity [66], alephium [147], Qubic [73], and Plasma [109] enable general-purpose computation on a BlockDAG by way of smart contracts. However, they ultimately rely on a single global consensus mechanism for the integrity of every application.

Sharded blockchains, including Omniledger [77], Elastico [93], RapidChain [152], RSCoin [43], and Ethereum 2.0 [25] are a form of BlockDAG. Most still require that all applications have essentially the same trust assumptions.

Other sharded blockchain projects, such as Aion [140], Cosmos [57], and Polkadot [150], envision heterogeneous chains with inter-chain communication. Polkadot features a single Relay Chain trusted by all parachains (parallelizable chains), although it does allow parachains to proxy for outside entities, including other blockchains. Perhaps most similarly to our multi-chain transactions (§3.6.3), Aion can use Bridges, consensus mechanisms trusted by multiple chains, to commit a transaction to each.

All of these blockchain projects operate at a higher level of abstraction than Charlotte. Charlotte is a generic format for communicating blocks, with a novel attestation-based model for specifying availability and integrity properties. However, we believe any of these projects could benefit from building their implementations within the Charlotte framework. For example, where Cosmos' Inter-Blockchain Communication [57] and Aion's Transwarp Conduits [68] require that one chain be able to read and validate transaction commits from another, we present a unified framework for the data they must request and interpret: integrity attestations.

3.9.3 Availability attestations

Although storage services are widely available [7, 62, 100], availability attestations (§3.3.7) make Wilbur servers unique. The only type of availability attestation we have implemented is a simple promise to store a block indefinitely. However, there is a great deal of work on reliable storage [49, 58] and proofs of retrievability [75, 23, 131] that could be used to make a variety of availability attestation subtypes that provide more availability with less trust.

3.9.4 Integrity attestations

Integrity attestations abstract over a variety of mechanisms lending integrity to data provenance and ADDS properties. In some ways, attestations resemble the labels of distributed information flow control systems [155, 91], and implement a kind of endorsement [154] as additional attestations are minted for the same block. In other ways, integrity attestations generalize ordering services for traditional distributed systems [71] or blockchains [139]. These services maintain a specific property of a ADDS (ordering), much like our blockchain integrity attestations. However, integrity attestations generalize over many possible properties: timestamps, provenance, etc.

Future integrity attestation subtypes might take advantage of technologies like authentication logic proofs and artifacts representing assurances of data provenance [8, 133].

3.10 Discussion

Charlotte offers a decentralized framework for composable Authenticated Distributed Data Structures with well-defined availability and integrity properties. Together, these structures form the blockweb, a novel generalization of blockchains. Charlotte addresses many of the shortcomings of existing ADDSs by enabling parallelism and composability. Charlotte is flexible enough to enable applications patterned after any existing ADDS while offering rigorous guarantees through attestations that can be given precise semantics.

With Charlotte, heterogeneous observers can use heterogeneous applications across heterogeneous participants with well-defined failure tolarances. By embracing Least Ordering and heterogeneity, these new applications will save time and resources, and serve broader audiences.

CHAPTER 4

HETEROGENEOUS CONSENSUS

Synopsis

In distributed systems, a group of *observers* achieve *consensus* when, by observing the output of some *participants*, they all arrive at the same value. Consensus is crucial for ordering transactions in failure-tolerant systems. Traditional consensus algorithms are homogeneous in three ways: all observers are created equal, all participants are created equal, and all failures are created equal. We present the first consensus algorithm to be heterogeneous in all three respects. Each observer sets their own assumptions concerning differently trusted sets of participants, and which mixed failures to tolerate. We express these assumptions in a novel *observer Graph*, and demonstrate exactly when consensus is possible.

We present *Heterogeneous Consensus*: an extension of Byzantine Paxos. Heterogeneous Consensus can achieve consensus for any viable Observer Graph in best-case three message sends. Heterogeneous Consensus is ideal for federated systems such as blockchains, in which parties make different trust assumptions. We also present a proof-of-concept implementation, which uses Charlotte (chapter 3) to construct composable blockchains with heterogeneous trust.

4.1 Introduction

Consensus is a classic distributed systems problem in which observers[1] (§ 1.1) try to decide on the same value, based on the outputs of some set of *participants*, some of whom may fail. It is a vital part of any fault-tolerant system maintaining strongly consistent state, such as Datastores [39, 27] or Blockchains [106, 56, 40], or indeed anything which orders transactions. Traditionally, consensus algorithms have been *homogeneous* in three ways:

- All observers are created equal. All observers make the same assumptions, and so system guarantees apply either to all observers, or none.
- All participants are created equal. Traditionally, systems tolerate at most some number f of failed participants. However, it doesn't matter which f fail. One participant is "just as good as" another.
- All failures are created equal. Systems are traditionally designed to tolerate either byzantine or crash failures. There is therefore no distinction between different failure scenarios in which the same participants fail, but possibly in different ways.

As a result, traditional consensus is ill-suited to federated systems, in which different parties make different assumptions about who to trust, and when. We present the first *heterogeneous* consensus algorithm, which makes none of these assumptions.

Heterogeneity allows participants to tailor a consensus protocol for the specific requirements of observers, rather than trying to force everyone to agree when-

[1] Lamport calls them *learners*[84].

ever any pair demand to agree. This can save time and resources, or even make consensus possible where it was not before (§ 4.8).

We go on to implement this algorithm in Fern servers using Charlotte (§ 3.4.2). With this implementation, we can build composable blockchains with heterogeneous trust, allowing application-specific chains to order only the blocks they need (§ 4.9.1).

4.2 Consensus

Intuitively, the purpose of any consensus is for all the observers to decide on one and only one value, and for them all to decide on the same value. Consensus is distinct from *agreement* [108] in that when multiple different values are proposed, agreement algorithms may get stuck, and observers may never decide.

Here, we use *round* to refer to a specific execution: the actions of a specific set of participants during some specific timeframe. *Protocol* refers to the instructions correct participants follow in a round, and *algorithm* refers to the mathematical construct used to create a protocol for a specific set of participants. In any consensus, a *round* begins when participants *propose* candidate values. After receiving some messages from participants, each observer eventually *decides* on a single value. For example, Alice, Bob, and Carol may want to use the Paxos consensus algorithm [83] to decide what to eat for lunch. They agree to follow a protocol, specified by the Paxos algorithm, for their specific set of participants and failure tolerances. Then they execute the protocol, completing a round of consensus. The same participants could run another round with the same protocol, if they want to decide on, say, what to eat for supper.

Traditionally, consensus is expressed in terms of 4 formal properties: Non-Triviality, Integrity, Agreement, and Termination.

4.2.1 Non-Triviality

Intuitively, a consensus protocol shouldn't allow observers to simply always decide some pre-determined value. For a round of consensus to be non-trivial, no correct any value unless that value has been proposed. A consensus protocol is non-trivial if all possible executions result in non-trivial rounds of consensus. Non-triviality is the same in heterogeneous and homogeneous settings.

Our Heterogeneous Consensus algorithm ensures non-triviality: observers do not decide a value unless they have verified that a participant has signed a proposal for that value.

4.2.2 Integrity

Intuitively, a consensus protocol shouldn't allow the same observer to decide different things. An observer in a round of consensus has integrity if it decides at most one distinct value. A homogeneous consensus protocol has integrity if, whenever its failure assumptions are correct, all possible executions result in rounds of consensus in which all observers have integrity.

Generalizing integrity from the homogeneous setting into a heterogeneous one is deceptively difficult. For example, in our prior work on Heterogeneous Fast Consensus, we distinguish between "gurus," observers whose failure assumptions are correct, and "chumps," who hold incorrect assumptions [130]. Similarly, Stellar

calls them "intact" and "befouled," respectively [96]. A Heterogeneous Consensus protocol is specified in terms of the exact (and possibly different) conditions under which each observer is guaranteed integrity (§ 4.3).

4.2.3 Termination

Intuitively, the round of consensus eventually ends, and every observer eventually decides. We can think of an observer as *terminating* or *finishing* when it decides. A homogeneous consensus protocol has termination if, whenever its failure assumptions are correct, all possible rounds result in all correct observers deciding after finite time. Various protocols measure "finite time" differently, and rely on different assumptions to guarantee (sometimes probabilistic) termination [54, 84, 103].

Like integrity, generalizing termination from the homogeneous setting into a heterogeneous one can be difficult. In prior work, such as Stellar [96], observers that maintain integrity also decide. Our failure assumptions can be more nuanced, and so we distinguish between which servers are guaranteed integrity, and which are guaranteed termination.

4.2.4 Agreement

Intuitively, all observers should decide the same value. In a round of consensus, two observers *agree* if at least one of them never decides, or if they have both decided the same value. A homogeneous consensus protocol has Agreement if, whenever its failure assumptions are correct, all possible executions eventually result in all pairs of correct observers agreeing.

Our generalization of Agreement from the homogeneous setting to a heterogeneous one is the key insight that makes our conception of Heterogeneous Consensus possible. For a given Heterogeneous Consensus protocol instance, we calculate the precise conditions under which each pair of observers must agree. This generalizes not only the homogeneous approach, but also the "intact nodes" concept from Stellar [96], and "linked nodes" from Cobalt [94].

4.3 The Observer Graph

We characterize observers' failure assumptions with a novel construct called an *observer graph*. In an observer graph, vertices are observers, and each pair of observers is connected by an edge, labeled with the conditions under which those observers must agree (§ 4.2.4). This generalizes Stellar's "slices" [96] and Cobalt's "essential sets" [94].

4.3.1 Universes

We specify these conditions as a set of *universes*. Each universe features a distinct set of failures. In particular, we model each universe as a pair, featuring the set of participants which are *safe* (act only as specified by the protocol), and the set of participants which are *live* (eventually send messages). The complements of these sets are the *byzantine* [82] and *crash* [87] failures, respectively, in each universe. Note that we assume byzantine participants can also fail to send any messages, so all byzantine failures are also crash failures.

Thus each edge in an observer graph is labeled with a set of universes, each of which is a pair of sets of participants. Thus for observers a and b, the set of universes in which they want to agree is written:

$$a\text{–}b$$

We illustrate example values for observer graph edge labels in Fig. 4.2. If two observers don't require agreement under any conditions, then the label on the edge between them is simply the empty set.

We also define a specific universe, REALITY, representing the set of participants that are actually safe (non-byzantine), and the set that are actually live (non-crashed).

Assumptions We assume that a byzantine participant can choose not to send any messages, so if the universe $\langle s, \ell \rangle$ is in $a\text{–}b$ (where s is a set of safe participants, and ℓ is a set of live participants), then $\langle s, s \cap \ell \rangle$ must be as well:

Def. 9.

$$\langle s, \ell \rangle \in a\text{–}b \Rightarrow \langle s, s \cap \ell \rangle \in a\text{–}b$$

Furthermore, we also assume that a strict subset of failures is always tolerated:

Def. 10. *Subset of failures property:*

$$\forall x, y. \langle s, \ell \rangle \in a\text{–}b \Rightarrow \langle s \cup x, \ell \cup y \rangle \in a\text{–}b$$

4.3.2 Example

For example, consider a situation in which observers might agree that they want to tolerate 1 byzantine failure out of 4 participants, but disagree about who the 4

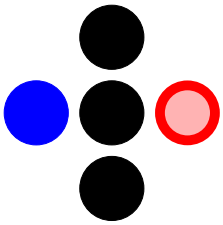

Figure 4.1: Observer Graph Example Scenario: Here, we draw one the blue participant as a solid blue circle, the red participant as a hollow red circle, and the black participants as black circles.

---

relevant participants are. Suppose there are 4 observers: 2 red and 2 blue, as well as 5 participants: 1 red, 1 blue, and 3 black. The participants are illustrated in Fig. 4.1.

The red observers want to agree if there is at most 1 byzantine failure amongst the red and black participants, even if the blue participant has failed. Likewise, the blue observers want to agree if there is at most 1 byzantine failure amongst the blue and black participants, even if the red participant has failed. The red observers and blue observers acknowledge that they may disagree with each other if a black participant fails, but otherwise they want to agree. Thus we draw the observer graph in Fig. 4.2.

4.3.3 Undirected

Agreement is also undirected: It makes no sense for a to agree with b when b disagrees with a. Put another way, if a agrees with b, then b agrees with a. Edges in the observer graph are likewise undirected:

$$a\text{–}b = b\text{–}a$$

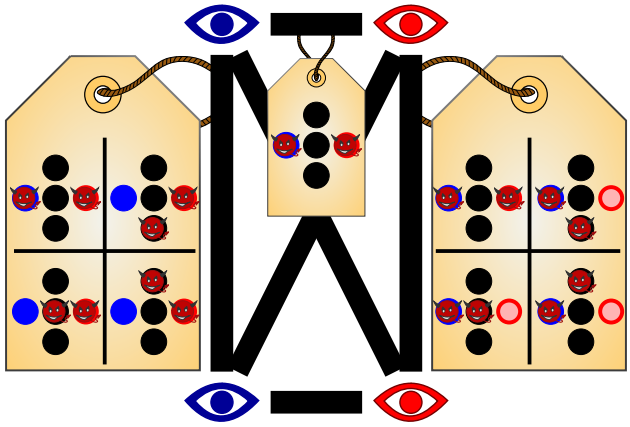

Figure 4.2: Example Observer Graph: Observers are drawn as eyes, with darker blue observers on the left, and lighter, outlined red observers on the right. For each pair of observers a and b, we show the value a–b as a label on the edge between them. In each universe shown within a label, byzantine failures are marked with a demonic face. All the edges except the rightmost and leftmost share the same label, in the middle.

4.3.4 Transitivity

Among observers with integrity (§ 4.2.2), agreement (§ 4.2.4) is transitive: if a agrees with b and b agrees with c, then a agrees with c. As a result, a and c must agree whenever both the conditions a–b and b–c are met. When observers' requirements reflect this assumption, we call the resulting observer graph *condensed*.

4.3.5 Condensed Observer Graph (COG)

A *Condensed Observer Graph* (*COG*) represents all the conditions under which each pair of observers must actually agree, after transitivity is taken into account. As a result of the transitive property of agreement, for any observer graph G featuring principals a, b, and c, we can create a new observer graph G', in which

a–c is replaced by:

$$a\text{–}c \cup (a\text{–}b \cap b\text{–}c)$$

Starting with observer graph G, and applying this operation repeatedly results in a Condensed Observer Graph (COG): an observer graph in which this operation doesn't change anything when applied to any pair of adjacent edges.[2] Equivalently, an observer graph G is *condensed* iff:

Def. 11. *Condensed*

$$\forall a, b, c.\ a\text{–}c \supseteq (a\text{–}b \cap b\text{–}c)$$

Given Def. 10, we can formulate an equivalent expression in terms of safe and live sets:

Lemma 5.

$$(a\text{–}c \supseteq (a\text{–}b \cap b\text{–}c)) \Leftrightarrow (\langle s, \ell\rangle \in a\text{–}b \wedge \langle s', \ell'\rangle \in b\text{–}c \Rightarrow \langle s \cup s', \ell \cup \ell'\rangle \in a\text{–}c)$$

Proof. First, we show:

$$(a\text{–}c \supseteq (a\text{–}b \cap b\text{–}c)) \Rightarrow (\langle s, \ell\rangle \in a\text{–}b \wedge \langle s', \ell'\rangle \in b\text{–}c \Rightarrow \langle s \cup s', \ell \cup \ell'\rangle \in a\text{–}c)$$

Given $(a\text{–}c = a\text{–}c \cup (a\text{–}b \cap b\text{–}c)) \wedge \langle s, \ell\rangle \in a\text{–}b \wedge \langle s', \ell'\rangle \in b\text{–}c$, by Def. 10:

$$\begin{aligned} & \langle s \cup s', \ell \cup \ell'\rangle \in a\text{–}b \\ \wedge\ & \langle s \cup s', \ell \cup \ell'\rangle \in b\text{–}c \end{aligned}$$

Therefore, since $a\text{–}c \supseteq (a\text{–}b \cap b\text{–}c)$:

$$\langle s \cup s', \ell \cup \ell'\rangle \in a\text{–}c$$

[2] With the Floyd-Warshall algorithm [55], this can be done in $O\left(|G|^3\right)$ time.

Second, we show

$$(a\text{–}c \supseteq (a\text{–}b \cap b\text{–}c)) \Leftarrow (\langle s, \ell \rangle \in a\text{–}b \wedge \langle s', \ell' \rangle \in b\text{–}c \Rightarrow \langle s \cup s', \ell \cup \ell' \rangle \in a\text{–}c)$$

For any $\langle s, \ell \rangle \in a\text{–}b \cap b\text{–}c$, it certainly holds that

$$\begin{aligned} & \langle s, \ell \rangle \in a\text{–}b \\ \wedge \quad & \langle s, \ell \rangle \in b\text{–}c \end{aligned}$$

Given $\langle s, \ell \rangle \in a\text{–}b \wedge \langle s', \ell' \rangle \in b\text{–}c \Rightarrow \langle s \cup s', \ell \cup \ell' \rangle \in a\text{–}c$, it follows that $a\text{–}b \cap b\text{–}c \supseteq a\text{–}c$.

□

Self-Edges

With a COG, we can define when an observer a will agree with itself (have Integrity § 4.2.2) simply as $a\text{–}a$. As one might expect, an observer must agree with itself if it's going to agree with anyone else:

Lemma 6.

$$a\text{–}b \subseteq a\text{–}a$$

Proof. follows from the definition of Condensed (Def. 11), and the fact that the COG is undirected (§ 4.3.3) □

4.4 Byzantine Paxos Variant

Heterogeneous Consensus is built on a traditional (meaning homogeneous, as opposed to heterogeneous) byzantine Paxos variant [85]. This protocol is conceptu-

ally much simpler than the popular *Practical Byzantine Fault Tolerance* [29]. In Lamport's terminology, our variant uses a simulated leader.

4.4.1 Assumptions and Definitions

- **Participants** are servers which both send and receive messages as part of the protocol. These are equivalent to Lamport's *acceptors* [85].
- **Clients** are machines that propose potential values for consensus. These are similar to Lamport's *proposers* [85].
- **Observers** receive messages from the participants. Observers *decide* on values. These are Lamport's *learners* [85].
- Participants can send **messages** to each other over the network, and **digital signatures** make messages unforgeable.
- Messages can **reference** other messages *by collision-resistant hash*: if one message contains a hash of another, it uniquely identifies the message it is referencing [112].
- **Safe** participants act only as specified by the protocol.
- **Live** participants eventually send messages.
- **Correct** participants are safe and live.
- **Faulty** participants are not correct.
- Any message from one correct participant to another correct participant eventually arrives.
- We use Q to designate a set of **quorums**, each of which is a set of participants.

- Agreement (§ 4.2.4) is only guaranteed so long as all pairs of quorums have at least one correct participant in their intersection.
- Termination (§ 4.2.3) is guaranteed only so long as at least one quorum is correct, and under specific network conditions (see Def. 31.

Our Consensus protocols are designed to satisfy the objectives laid out in § 4.2.

4.4.2 The Algorithm

Our byzantine Paxos variant can be thought of as taking place in *stages*, which are identified by *ballots*, to borrow Lamport's terminology [83]. Ballots can be thought of as natural numbers, and generally increase with time. We'll go over how ballot numbers are generated later.

We assume all correct participants echo all messages to all other correct participants. This ensures that if one correct participant receives a message, all correct participants receive that message.

Finally, we assume all participants ignore all messages for any ballot except the highest they've seen. It may be possible to optimize an implementation with additional messages pertaining to outdated ballots, but these are not strictly necessary for correctness. For instance, a participant might inform a proposer that their proposal has been ignored because of its ballot number, and the proposer might then know to try again later.

For Ballot b

The actions in each ballot are broken down into three *phases*, traditionally called *Phase 1a*, *Phase 1b*, and *Phase 2a* [84]. Each has a corresponding subtype of message.

1a: A client proposes a value v by creating a new ballot number b, and sending a *1a* message containing v and b to all participants.

1b: Upon receiving a *1a* message, a participant sends a *1b* message, which contains the *1a*, and the *2a* message with the highest ballot number that the participant has ever seen (if any). In homogeneous byzantine Paxos, a *1b*, only the highest ballot *2a* message matters, so the *1b* message need not include any others.

We say that a *1b* has *value* v' if it contains a *2a* with value v', or it contains no *2a*, and references a *1a* which contains v'.

We say that a *1b* has *ballot* b if it contains a *1a* which contains b.

2a: If a participant receives *1b* messages all with ballot b and value v', signed by a quorum of participants, it generates a *2a* message, referencing all the *1b* messages with b and v', which is said to have value v'. This is equivalent to Lamport's *accepting a value* [85]. It then sends the *2a* message to all observers.

If an observer receives *2a* messages all with ballot b and value v', signed by a quorum of participants, it *decides* value v'.

Multiple Ballots

It is possible for a ballot to *fail*: after some number of ballots, it may be the case that all messages have arrived, the protocol above doesn't require any participant to send any further messages, and yet no observer has yet decided. For this reason, it is necessary to start a new ballot when it is clear an old one isn't going well.

One way to handle this is to leave the responsibility at the clients: if a client proposes a ballot, and observers don't decide for a while, then the client should propose again. Randomized exponential backoff can be used to allow clients to adapt to the unknown delay in a semi-synchronous network without flooding the system.

Another technique is to have participants propose after a ballot has failed: If a participant assembles a quorum of *1b* messages in ballot b, and they *do not* all feature the same value, and it hasn't seen any higher ballot, then it proposes a new ballot, using one of the values from one of the *1b* messages it has seen. Likewise, *2a* messages could be sent to participants, so that if a participant assembles a quorum of *2a* messages in ballot b, and they *do not* all feature the same value, and it hasn't seen any higher ballot, it proposes a new ballot, using one of the values from one of the *2a* messages it has seen.

4.5 Heterogeneous Consensus

Heterogeneous Consensus[3] is a consensus algorithm (§ 4.2) based on Leslie Lamport's byzantine-fault-tolerant [82] variant of Paxos [83, 84] as described in § 4.4.

4.5.1 Heterogeneity

Most consensus protocols are *homogeneous* in 3 ways:

- Homogeneous *failures*: Protocols tend to tolerate either crash [87] or byzantine [82] failures. Traditional protocol and algorithm descriptors include "byzantine-fault-tolerant" and "crash-fault-tolerant."
- Homogeneous *participants*: Usually, protocols aren't specified in terms of *which* participants can fail, just *how many*. For instance, we often say that a protocol tolerates f failures out of n participants. Any f; all participants are the same.
- Homogeneous *observers*: Like most consensus protocols, our byzantine Paxos variant (§ 4.4) assumes a set of quorums, Q, which is universal, and provides no guarantees if failures violate certain assumptions about those quorums. For instance, it assumes that there will always be a correct participant in the intersection of any two quorums.

Heterogeneous Consensus is different in 3 ways:

[3] Heterogeneous Consensus is *a different algorithm* from an earlier project with a similar name, *Heterogeneous Fast Consenus* [130], which solves the related *fast consensus* [138] problem: it tolerates fewer failures, but has faster best-case decision latency than regular consensus.

- Heterogeneous *failures*: A homogeneous protocol which can tolerate 2 byzantine failures can tolerate 1 byzantine and 1 crash. Some protocols, however, are able to tolerate 1 byzantine and one crash, but not 2 byzantine. Such a protocol has a *mixed failure model* [134]: it is neither a homogeneous byzantine-fault-tolerant protocol, nor a homogeneous crash-fault-tolerant protocol.
- Heterogeneous *participants*: We might imagine a system in which we expect that *either* Alice and Bob fail, *or* Carol fails, but not Alice and Carol. That would not be homogeneous.
- Heterogeneous *observers*: We might imagine a world in which different observers have different opinions about what failures are possible. They might have different quorums, and different assumptions. Some observers might assume incorrectly, while others might assume correctly.

4.5.2 Assumptions and Definitions

Most of our assumptions are the same as in the homogeneous case (§ 4.4.1). In order to make this section (§ 4.5) a complete description, we repeat them here:

- **Participants** are servers which both send and receive messages as part of the protocol. These are equivalent to Lamport's *acceptors* [85].
- **Clients** are machines which propose potential values for consensus. These are similar to Lamport's *proposers* [85].
- **Observers** receive messages from the participants. Observers *decide* on values. These are Lamport's *learners* [85].

- Participants can send **messages** to each other over the network, and **digital signatures** make messages unforgeable.
- Messages can **reference** other messages *by collision-resistant hash*: if one message contains a hash of another, it uniquely identifies the message it is referencing [112].
- **Safe** participants act only as specified by the protocol.
- **Live** participants eventually send messages.
- **Correct** participants are safe and live.
- **Faulty** participants are not correct.
- Any message from one correct participant to another correct participant eventually arrives.

We make two additional assumptions in the Heterogeneous case, which are not in the homogeneous one:

- Each observer a has a set of quorums Q_a, each of which is a set of participants.
- The quorums are valid for the Condensed Observer Graph (§ 4.3.5).

4.5.3 Valid Quorums

The notion of quorums being *valid* for a COG replaces the homogeneous assumption that every pair of quorums have a non-byzantine intersection. We assume that each observer a has a set of quorums Q_a, each of which is a set of participants. However, not all choices of quorums are compatible for all *COG*s (§ 4.3.5). Recall that in the homogeneous case, we assumed:

- Agreement (§ 4.2.4) is only guaranteed so long as all pairs of quorums have at least one correct participant in their intersection.
- Termination (§ 4.2.3) is only guaranteed so long as all participants in at least one quorum are correct, and under specific network conditions (Def. 31).

There are similar requirements for the heterogeneous case. Recall that REALITY is a specific universe representing the set of participants that are actually safe and the set that are actually live (§ 4.3.1).

Liveness Requirement

All universes in which an observer a wants to agree with itself (or any other observer, by Lemma 6) feature at least one live quorum. We say an *observer* is live if one of its quorums is live:

Def. 12.

$$Live(x) \triangleq \langle \_, \ell \rangle = REALITY \wedge \ell \in Q_a$$

Under certain network conditions (Def. 31), live observers will eventually decide.

Safety Requirement

In all universes in which observers a and b want to agree, all of their live quorums must intersect on at least on safe participant.

Def. 13.

$$\left(\begin{array}{ll} & \langle s, \ell \rangle \in a\text{–}b \\ \wedge & q_a \in Q_a \\ \wedge & q_b \in Q_b \\ \wedge & q_a \subseteq \ell \\ \wedge & q_b \subseteq \ell \end{array}\right) \Rightarrow q_a \cap q_b \cap s \neq \emptyset$$

Picking Good Quorums

Given a COG, a safe choice of quorums would be any live set from any edge to which the observer connects:

$$Q_a = \left\{ \ell \;\middle|\; \langle s, \ell \rangle \in a\text{–}b \in COG \right\}$$

4.5.4 Heterogeneous Objectives

Intuitively, the purpose of any consensus is still for all the observers to decide on one and only one value, and for them all to decide on the same value. However, we now face the challenge of redefining our objectives, given that the *real* universe, which is to say the set of failures that actually happen, may not be in all the edges of the COG: not all observers may end up agreeing.

Entangled

We say two observers are entangled if their failure assumptions correctly anticipated the set of failures that actually happen:

Def. 14.

$$Entangled(a, b) \triangleq REALITY \in a\text{–}b$$

Entangled observers are guaranteed agreement (§ 4.2.4).

Correct

We say an *observer* is correct if it is live and entangled with itself.

Def. 15.

$$Correct(a) \triangleq Live(a) \wedge Entangled(a, a)$$

The formal requirements of Heterogeneous Consensus are as follows (differences from heterogeneous consensus are in blue):

- **Termination**: Every correct observer decides some value (§ 4.2.3).
- **Non-Triviality**: No observer can decide on any value unless that value has been proposed (§ 4.2.1).
- **Integrity**: Every correct observer decides at most one value (§ 4.2.2).
- **Agreement**: If 2 entangled observers each decide, they decide the same value.

4.5.5 Key Idea

Conceptually, within each round of Heterogeneous Consensus, we run a round of byzantine Paxos for each observer, but allow *2a* data from *other instances* in *1b* messages. The *2a* from other instances means that each instance won't decide

contrary to the other ones, unless there are byzantine failures which would allow the relevant observers to disagree. For efficiency, our instances share most of their messages. In fact, when all edges in the COG are the same, all messages are shared by all instances: Heterogeneous Consensus reduces to byzantine Paxos.

Intuitively, *1b* messages represent an attempt to gather a quorum of participants who have either never seen a value before, or can show that no different value can already have been decided. By including *2a* messages from other observers' rounds, we require that, before any observer can gather a quorum, it must show that the participants in that quorum show that no different value can already have been decided *by any observer*. There is an important exception: if a participant can be proven to have violated the protocol (a byzantine failure), then it may mean two observers do not have to agree, and they can ignore each other's *2a*s.

At a slightly more detailed level, each participant keeps track of *2a* messages it has formed for each observer, and considers an old *2a Buried* (Def. 27) only if it can be sure a quorum of participants have seen a different *2a* with a higher ballot. Each *1b* message, then, is only *valid* (Def. 30) if all the sender's unburied *2a* messages have the same value as the *1b* message. There is one crucial exception: byzantine participants can be *caught* (Def. 24) disobeying the protocol, and demonstrate that observers are definitely not entangled (Def. 14), in which case 2as for one observer will not affect *1b* messages for the other.

There are a couple of other differences, which make Heterogeneous Consensus easier to analyze:

- All messages have the same ballot and value as the "most recent" *1a* message.
- All messages (transitively) reference all prior messages.

- Participants can only form a *2a* message if they sent one of the *1b* messages that goes into it.

4.5.6 Messaging

Participants send messages to each other. We assume a message between correct participants sent eventually arrives. Correct participants echo all messages sent and received to all other participants, ensuring that if one correct participant receives a message, all correct participants eventually receive it. When safe participants receive a message, they and send any resulting messages (specified by the protocol) atomically: they do not receive more messages between sending results to various other participants. In addition, when safe participants receive a message, and they've already received a message with a higher ballot number, they retain the message (reference it in future messages), but do not act on it.

Each message x contains some kind of cryptographic signature allowing anyone to identify the signer, or signator, written $\text{Sig}(x)$

Def. 16.

$$Sig(x : message) \triangleq \textit{ the participant that signed } x$$

We can define Sig over sets, to mean the set of signators of the messages:

Def. 17.

$$Sig(x : set) \triangleq \left\{ Sig(m) \;\middle|\; m \in x \right\}$$

Furthermore, each message x carries references to 0 or more other messages, $x.refs$. These references are by hash, ensuring no cycles in the reference graph, and

that it's possible to know exactly when one message references another [112]. Correct participants delay receipt of any message until they have received all messages it references. This ensures they receive, for example, a $1a$ for a given ballot before receiving any $1b$s for that ballot. Whenever a correct participant sends a message, it includes in that message's references the most recent message it has sent, and any messages it received since sending it.

Each message also has a unique identifier, and an identifiable type: $1a, 1b$, or $2a$.

1a

A $1a$ message x has 2 type-specific fields:

- $x.value$ is a proposed value. We assume these are valid (invalid values must be ignored).
- $x.ballot$ is a natural number specific to this proposal.

We assume that:

Def. 18.

$$x.ballot = y.ballot \Rightarrow x = y$$

This can be accomplished by including signature information in the least significant, say, 256 bits of the ballot number.

1b

A $1b$ message has no type-specific fields. A correct participant sends a $1b$ message when it receives a $1a$ message with a ballot number higher than any ballot number of any message it has ever received.

2a

A $2a$ message x has one type-specific field:

- $x.obs$ identifies a specific observer.

4.5.7 Decisions

An observer decides when it has observed a set of *2a* messages with the same ballot, sent by a quorum of participants. We call such a set a *decision*.

Def. 19.

$$Decision_a(q_a) \triangleq Sig(q_a) \in Q_a \land \forall \{x, y\} \subseteq q_a.\ b(x) = b(y) \land (x.obs = a) \land (x : 2a)$$

Although decisions are not messages, applications might send decisions in other messages as a kind of "proof of consensus." This is exactly how the Heterogeneous Consensus Integrity Attestations work in Charlotte (§ 4.9.1)

4.5.8 Machinery

In order to describe the Heterogeneous Consensus protocol, we require some mathematical machinery.

Transitive References

Each message x carries with it some set of references $x.refs$ to prior messages. We define Tran(x) to be the transitive closure of these references, so all the messages in the "causal past" of x.

Def. 20.

$$Tran(x) \triangleq \{x\} \cup \bigcup_{m \in x.refs} Tran(m)$$

Get1a

It is useful to refer to the 1a that started the ballot of a message. For a 1a this is itself, and for any other message it is simply the highest ballot number 1a in its transitive references.

Def. 21.

$$\begin{aligned}
Get1a(x : 1a) &\triangleq x \\
Get1a(x : 1b \text{ or } 2a) &\triangleq Get1a(Tran(x) - \{x\}) \\
Get1a(x : set) &\triangleq Get1a(m) \;\Big|\; \begin{array}{l} m \in x \\ \wedge \;\; \forall z \in x.\; b(Get1a(z)) \leq b(Get1a(m)) \end{array}
\end{aligned}$$

Ballot Numbers

The ballot number of a $1a$ is listed in a field of the message, and the ballot number of anything else is the highest ballot number among the messages it references.

Def. 22.

$$b(x) = Get1a(x).ballot$$

Value

Likewise, the value of a $1a$ is listed in a field of the message, and the value number of anything else is the value of the highest ballot $1a$ among the messages it references.

Def. 23.

$$V(x) = Get1a(x).value$$

Caught

Some behavior can create proof that a participant is byzantine. We say that a participant p is *Caught* in a message x if the transitive references of the messages include evidence such as two messages, m and m', both signed by p, in which neither is featured in the other's transitive references (correct participants reference all prior messages).

Def. 24.

$$Caught(x) \triangleq \left\{ Sig(m) \;\middle|\; \begin{array}{ll} & \{m, m'\} \subseteq Tran(x) \\ \wedge & Sig(m) = Sig(m') \\ \wedge & m \notin Tran(m') \\ \wedge & m' \notin Tran(m) \end{array} \right\}$$

4.5.9 Connected

When some participants are proven to be byzantine, it is clear that some observers are not connected in the COG, meaning there are no universes in the edge between them in which no "safe" participants are proven byzantine. It is clear that disconnected observers may not agree, and so messages which are "for" a specific observer (2a) will have some implications only for observers that are still connected.

Def. 25.

$$Con_a(x) \triangleq \left\{ b \;\middle|\; \begin{array}{ll} & \langle s, \_ \rangle \in a\text{–}b \in COG \\ \wedge & s \cap Caught(x) = \emptyset \end{array} \right\}$$

Quorums in Messages

Messages of type 2a reference quorums of messages with the same value and ballot. Note that the definition of these quorums depends on the definition of *valid*, which we define later (Def. 30).

A 2a's quorums are formed from valid 1b messages with the same ballot and value. However, the validity of a 1b depends on the observer.

Def. 26.

$$q(x : 2a) \triangleq \left\{ m \;\middle|\; \begin{array}{ll} & m : 1b \\ \wedge & valid_{x.obs}(m) \\ \wedge & m \in Tran(x) \\ \wedge & b(m) = b(x) \end{array} \right\}$$

Buried

A 2a can become irrelevant if, after a time, an entire quorum have seen 2as with different values and higher ballot numbers. We call such a 2a *Buried* (in the context of some later message):

Def. 27.

$$Buried(x : 2a, y) \triangleq \left\{ Sig(m) \;\middle|\; \begin{array}{ll} & m \in Tran(y) \\ \wedge & z : 2a \\ \wedge & \{x, z\} \subseteq Tran(m) \\ \wedge & z.obs = x.obs \\ \wedge & V(z) \neq V(x) \\ \wedge & b(z) > b(x) \end{array} \right\} \in Q_{x.obs}$$

Well-Formedness

No 2a should have an invalid quorum upon creation. Furthermore, no participant should create a 2a unless it sent one of the 1bs in the 2a.

Similarly, no 1b should reference any other message with the same ballot number besides a 1a (correct participants make 1bs as soon as they receive a 1a). Participants should ignore messages that are not well formed.

Def. 28.

$$\begin{array}{ll} & \forall x : 1b, y.\ (y \in Tran(x)) \ \wedge\ (x \neq y \neq Get1a(x)) \ \Rightarrow\ (b(y) \neq b(x)) \\ \wedge & \forall x : 2a.\ q(x) \in Q_{x.obs} \ \wedge\ Sig(x) \in Sig(q(x)) \end{array}$$

Connected 2as

Entangled observers must agree, but observers that are not connected (as of any message) are not entangled, so they need not agree. A *1b* message includes a *2a* message, in a sense, to demonstrate that some observer may have decided some value. For observer a, it can be useful to find the set of *2a* messages from the same sender as a message x (and sent earlier) which are still unburied, and for observers connected to a. The *1b* should not be used to make any new *2a* messages for observer a that have values different from these *2a* messages.

Def. 29.

$$Con2as_a(x) \triangleq \left\{ m \;\middle|\; \begin{array}{ll} & m : 2a \\ \wedge & m \in Tran(x) \\ \wedge & Sig(m) = Sig(x) \\ \wedge & m.obs \in Con_a(x) \\ \wedge & \neg Buried(m, x) \end{array} \right\}$$

Valid 1bs

Participants send a 1b message whenever they receive a 1a message with a higher ballot number than they've yet seen. However, this does not mean that the 1b's value (which is the same as the 1a's) will agree with that of 2as the participant has already sent. We call a 1b message *valid* (with respect to an observer) when its value agrees with that of unburied 2as the participant has sent.

Def. 30.

$$valid_a(x : 1b) \triangleq \forall m \in Con2as_a(x).\ V(x) = V(m)$$

4.5.10 The Heterogeneous Consensus Protocol

Like our byzantine Paxos variant (§ 4.4), Heterogeneous Consensus can be thought of as taking place in *steps*, which are called (or identified by) *ballots*, to borrow Lamport's terminology [83]. Ballots can be thought of as natural numbers, and generally increase with time (§ 4.5.10.

Differences from the homogeneous protocol (§ 4.4) are in blue.

For Ballot b

The actions in each ballot are broken down into three *phases*, traditionally called *1a*, *1b*, and *2a* [84].

1a: A client proposes a value v by creating a new ballot number b, and sending a *1a* message containing v and b to all participants.

1b: Upon receiving a *1a*, a participant sends a *1b*, which references all messages it has ever received (transitively).

2b: If a participant receives a *1b* message m, and has never received a message with a higher ballot number, it creates a 2a m' for each observer, and if m' is well-formed (Def. 28), and if $m \in \mathrm{q}(m'.obs)$, it sends m' to all participants.

If an observer a receives *2a* messages for observer a all with ballot b and value v', signed by one of observer a's quorums of participants, then observer a decides value v'.

Multiple Ballots

Just as in the homogeneous case, it is possible for a ballot to *fail*: after some number of ballots, it may be the case that all the participants are done, and no one has yet decided. For this reason, it is necessary to start a new ballot when it is clear an old one isn't going well.

One way to handle this is to leave the responsibility at the clients: if a client proposes a ballot, and observers don't decide for a while, then the client should propose again. Randomized exponential backoff can be used to allow clients to adapt to the unknown delay in a semi-synchronous network without flooding the system.

Another way is to have participants propose after a ballot has failed: when sufficiently many 1b messages for a given ballot are collected, but they are all invalid, a participant could send a new 1a. There are subtleties to ensuring liveness, which we discuss in § 4.6.7.

Ballot Numbers

There are well-established schemes for ensuring a unique ballot number for each proposal, and making these generally increase over time. In our implementation, ballot numbers are lexicographically ordered pairs featuring the current time, and the client signature of the hash of the value for that ballot. The latter ensures no two *1a*s with the same ballot have different values. The former ensures that ballot numbers generally increase with time.

To prevent clients from entering artificially high ballot numbers (which could be undesirable), each participant delays the receipt of any received message until its own clock exceeds the message's ballot number's time. If all clocks are synchronized, the "best" a client can do (in some sense) is to use the correct time.

4.6 Correctness

4.6.1 Useful Lemmas

First, we build up some useful facts about Heterogeneous Consensus.

For example, any two messages with the same ballot have the same 1a, and thus the same value.

Lemma 7.

$$b(x) = b(y) \Rightarrow \mathit{Get1a}(x) = \mathit{Get1a}(y) \wedge V(x) = V(y)$$

Proof. Follows from the ballot assumption (Def. 18), and the definition of $\mathit{Get1A}$ (Def. 21) and V (Defnition Def. 23). □

Con() and Caught()

No correct participant can be caught in any message.

Lemma 8.

$$p \textit{ is correct } \Rightarrow p \notin \mathit{Caught}(x)$$

Proof. By the definitions of correct behavior, the unforgeability of signatures, and the definition of Caught (Def. 24). □

Strictly later messages always catch a superset of participants compared to earlier messages.

Lemma 9.

$$m \in \mathit{Tran}(x) \Rightarrow \mathit{Caught}(m) \subseteq \mathit{Caught}(x)$$

Proof. Follows from the definition of Caught (Def. 24) and Tran (Def. 20). □

As a result, strictly later messages connect subsets of observers. In a sense, once observers are disconnected, they can't reconnect: the failures have already occurred.

Lemma 10.

$$m \in \mathit{Tran}(x) \Rightarrow \mathit{Con}_a(x) \subseteq \mathit{Con}_a(m)$$

Proof. Follows from the definition of Con (Def. 25) and Lemma 9. □

Con() is actually a partial equivalence relation. That means, as of some message x, if a is connected to b, then b is connected to the same set of observers as a.

Lemma 11.

$$b \in \mathit{Con}_a(x) \Rightarrow \mathit{Con}_b(x) = \mathit{Con}_a(x)$$

Proof. Follows from the definition of COG and Con (Def. 25). □

Furthermore, as of message x, if a is connected to b, then b is connected to a.

Lemma 12.

$$a \in \mathit{Con}_b(x) \Rightarrow b \in \mathit{Con}_a(x)$$

Proof. By the definition of Con (Def. 25),

$$\exists \langle s, \ell \rangle \in b\text{–}a \in \mathrm{COG} \ \wedge \ s \cap \mathrm{Caught}(x) = \emptyset$$

COG is undirected, so $a\text{–}b = b\text{–}a$.

$$\Rightarrow \langle s, \ell \rangle \in a\text{–}b \in \mathrm{COG}$$

$$\Rightarrow b \in \mathrm{Con}_a(x)$$

□

As a result, if an observer is connected to anyone, it must be connected to itself.

Lemma 13.

$$\mathit{Con}_a(x) \neq \emptyset \Rightarrow a \in \mathit{Con}_a(x)$$

Proof. By Lemma 12 and Lemma 11. □

If a message x references two quorums' messages, and the observers for those quorums are still connected as of x, then there must be an uncaught participant who signed at least one message in each.

Lemma 14.

$$\left(\begin{array}{ll} & \mathit{Sig}(q_a) \in Q_a \\ \wedge & \mathit{Sig}(q_b) \in Q_b \\ \wedge & q_a \cup q_b \subseteq \mathit{Tran}(x) \\ \wedge & b \in \mathit{Con}_a(x) \end{array} \right) \Rightarrow \exists m_a \in q_a, m_b \in q_b, p \notin \mathit{Caught}(x).\ \mathit{Sig}(m_a) = \mathit{Sig}(m_b) = p$$

Proof. By the definition of Con (Def. 25):

$$\exists \langle s, \_ \rangle \in a\text{–}b \in \mathrm{COG} \wedge s \cap \mathrm{Caught}(x) = \emptyset$$

Therefore, by the definition of COG and quorum properties:

$$\mathrm{Sig}(q_a) \cap \mathrm{Sig}(q_b) \cap s \neq \emptyset$$

$$\therefore \exists p \in \mathrm{Sig}(q_a) \cap \mathrm{Sig}(q_b) \cap s$$

$$p \in \mathrm{Sig}(q_a) \Rightarrow \exists m_a \in q_a . \mathrm{Sig}(m_a) = p$$

$$p \in \mathrm{Sig}(q_b) \Rightarrow \exists m_v \in q_b . \mathrm{Sig}(m_b) = p$$

$$p \in s \Rightarrow p \notin \mathrm{Caught}(x)$$

□

4.6.2 1bs and 2as

The set of 2as connected to a given message is relative to the observer. However, connected observers will get the same set of 2as.

Lemma 15.

$$a \in \mathit{Con}_b(x) \Rightarrow \mathit{Con2as}_a(x) = \mathit{Con2as}_b(x)$$

Proof. By Lemma 11:

$$\mathrm{Con}_a(x) = \mathrm{Con}_b(x)$$

So, noting that $\mathrm{Buried}(x, y)$ is independent of the observer (Def. 27), by the definition of $\mathit{Con2as}$ (Def. 29):

$$\mathrm{Con2as}_a(x) = \mathrm{Con2as}_b(x)$$

□

It follows that if two observers are connected, a 1b that is valid for one of them will be valid for the other as well:

Lemma 16.

$$a \in \mathit{Con}_b(x) \Rightarrow \mathit{valid}_a(x) = \mathit{valid}_b(x)$$

Proof. Follows from Lemma 15, and the definition of *valid* (Def. 30). □

4.6.3 Entangled Observers

If two observers are entangled, then for all messages that are actually sent, they are connected.

Lemma 17.

$$\mathit{Entangled}(a, b) \Rightarrow a \in \mathit{Con}_b(x)$$

Proof. By the definition of Entangled (Def. 14):

$$\text{REALITY} = \langle s, \ell \rangle \in a\text{–}b$$

Here s represents the set of correct participants (Definition of REALITY). Therefore, by Lemma 8:

$$s \cap \text{Caught}(x) = \emptyset$$

So by the definition of *Con* (Def. 25):

$$a \in \text{Con}_b(x)$$

□

Entanglement is not really ordered.

Lemma 18.

$$\mathit{Entangled}(a, b) \Rightarrow \mathit{Entangled}(b, a)$$

Proof. By the definition of entangled (Def. 14):

$$\text{REALITY} \in a\text{–}b$$

COG is undirected, so $a\text{–}b = b\text{–}a$. Therefore:

$$\text{REALITY} \in b\text{–}a$$

And so:

$$\text{Entangled}(b, a)$$

□

Entanglement is also transitive. If a and b are entangled, and b and c are entangled, then so are a and c.

Lemma 19.

$$\mathit{Entangled}(a, b) \wedge \mathit{Entangled}(b, c) \Rightarrow \mathit{Entangled}(a, c)$$

Proof. Let $\langle s, \ell \rangle = \text{REALITY}$. By the definition of entangled (Def. 14):

$$\begin{aligned} & \langle s, \ell \rangle \in a\text{–}b \\ \wedge \quad & \langle s, \ell \rangle \in b\text{–}c \end{aligned}$$

By the definition of condensed (Def. 11):

$$\langle s \cup x, \ell \cup \ell \rangle \in a\text{–}c$$

$$\therefore \langle s, \ell \rangle \in a\text{–}c$$

And so by the definition of entangled (Def. 14):

$$\text{Entangled}(a, c)$$

□

It follows that if an observer is entangled to anyone, it must also be entangled with itself.

Lemma 20.

$$\mathit{Entangled}(a, b) \Rightarrow \mathit{Entangled}(a, a)$$

Proof. By Lemma 18:

$$\text{Entangled}(b, a)$$

And so by Lemma 19:

$$\text{Entangled}(a, a)$$

□

If an observer observes a quorum of 2as with the same ballot, then any 2as for an entangled observer with a higher ballot must have the same value.

Lemma 21.

$$\left(\begin{array}{ll} & \mathit{Entangled}(a, b) \\ \wedge & \mathit{Decision}_a(q_a) \\ \wedge & w : 2a \\ \wedge & w.obs = b \\ \wedge & b(w) > b(q_a) \end{array}\right) \Rightarrow V(q_a) = V(w)$$

Proof. As ballot numbers are natural numbers, we can prove this by **induction** on b(w).

Base Case: for $\mathrm{b}(w) \leq \mathrm{b}(q_a)$, Lemma 21 trivially holds.

Induction: Assume Lemma 21 holds for all values of $w < w'$. We now show that it holds for w'.

By well-formedness (Def. 28),

$$\mathrm{q}(w') \in Q_b$$

By Lemma 17:

$$b \in \mathrm{Con}_a(w')$$

By Lemma 14,

$$\exists m_a \in q_a, m_w \in \mathrm{q}(w'), p \notin \mathrm{Caught}(w').\mathrm{Sig}(m_a) = \mathrm{Sig}(m_b) = p$$

By the definition of Well-formed (Def. 28):

$$\forall (r : 2a) \in \mathrm{Tran}(m_w).\mathrm{b}(r) < \mathrm{b}(m_w)$$

By the definition of q (Def. 26), m_w is valid. By Lemma 17, $b \in \mathrm{Con}_a(w')$, and so by the definition of valid (Def. 30), either $m_a \in \mathrm{Con2as}_b(m_w)$, in which case:

$$\mathrm{V}(m_a) = \mathrm{V}(q_a) = \mathrm{V}(w') = \mathrm{V}(m_w)$$

or $\mathrm{Buried}(m_a, m_w)$. However, by the definition of buried (Def. 27):

$$\exists (r : 2a) \in \mathrm{Tran}(m_w).r.obs = a \wedge (\mathrm{b}(r) > \mathrm{b}(m_a)) \wedge (\mathrm{V}(r) \neq \mathrm{V}(m_a))$$

By Lemma 20, $\mathrm{Entangled}(a, a)$, so by our induction hypothesis, no such r exists. Therefore, m_a is not buried, so:

$$\mathrm{V}(q_a) = \mathrm{V}(w')$$

□

It follows that if an observer is entangled with anyone, and observes a quorum of 2as with the same ballot, then none of those 2as can ever be buried.

Lemma 22.

$$\mathit{Entangled}(a, b) \wedge \mathit{Decision}_a(q_a) \wedge x \in q_a \Rightarrow \neg \mathit{Buried}(x, w)$$

Proof. By the definition of Buried (Def. 27), there would have to exist a 2a for the same observer with a different value and a greater ballot number than b(q_a). By Lemma 20, a is entangled with itself, so by Lemma 21, no such 2a exists. □

4.6.4 Non-Triviality

No observer can decide on any value unless that value has been proposed (§ 4.2.1).

Theorem 7.

$$\mathit{Decision}_a(q_a) \Rightarrow \exists x : 1a.\, V(x) = V(q_a)$$

Proof. By the definition of Get1A for Decisions (Def. 21),

$$\text{Get1a}(q_a) : 1a \wedge \text{V}(\text{Get1a}(q_a)) = \text{V}(q_a)$$

□

4.6.5 Agreement

If two entangled observers (a and b) each decide, they decide the same value (§ 4.2.4).

Theorem 8.

$$\mathit{Entangled}(a, b) \wedge \mathit{Decision}_a(q_a) \wedge \mathit{Decision}_b(q_b) \Rightarrow V(q_a) = V(q_b)$$

Proof. If $\mathrm{b}(q_a) = \mathrm{b}(q_b)$, then by the ballot assumption (Def. 18, and definition of V (Def. 23): $\mathrm{V}(q_a) = \mathrm{V}(q_b)$.

Otherwise, without loss of generality, assume $\mathrm{b}(q_a) < \mathrm{b}(q_b)$. By the definition of decision (Def. 19):

$$\exists m : 2a \in q_b.\mathrm{V}(m) = \mathrm{V}(q_b)$$

By Lemma 21:

$$\mathrm{V}(m) = \mathrm{V}(q_b) = \mathrm{V}(q_a)$$

□

4.6.6 Integrity

An observer that is entangled with itself decides at most one value.

Theorem 9.

$$\mathit{Entangled}(a, a) \wedge \mathit{Decision}_a(q_a) \wedge \mathit{Decision}_a(q_b) \Rightarrow V(q_a) = V(q_b)$$

Proof. Follows directly from Agreement (Theorem Thm. 8). □

4.6.7 Termination

Consensus eventually terminates.

Network Assumption

Heterogeneous Consensus, and indeed our byzantine Paxos variant, rely on a very specific network assumption to guarantee termination. However, it is possible to use artificial delays to reduce other network assumptions to this one (§ 4.6.7).

Def. 31. *We assume:*

- *Eventually, there will be 13 consecutive periods of any duration, with no time in between, numbered 0 through 12, such that any message sent before one period begins is delivered before it ends.*
- *If a correct participant sends a message in between receiving two messages (m and m'), and m is delivered in some period n, then the message is sent in period n.*
- *No 1as will be delivered during any of the 13 periods except three: x , y, and z.*
- *x is delivered to a correct participant in period 0.*
- *y is delivered to a correct participant in period 4.*
- *z is delivered to a correct participant in period 9.*
- *$V(y) = V(z) =$ the value of the highest ballot 2a known to any correct participant at the end of period 3.*
- *$b(x) < b(y) < b(z)$.*
- *$b(x)$ is greater than any ballot number of any message delivered before period 0.*

These assumptions are *only necessary* for termination, not any safety property.

Proof

Theorem 10. *After period 12, if a is Correct (Def. 15), $\exists q_a.Decision_a(q_a)$*

Proof. Periods 0-3 are sufficient for all correct participants to send 1bs (and 2as) in response to any 1a delivered prior to period 0.

By the end of period 1, x will be delivered to all correct participants. They will therefore cease generating 2as for any ballot number $< \mathrm{b}(x)$.

By the end of period 3, all correct participants will have received all 1bs and 2as generated by correct participants with $\leq \mathrm{b}(x)$. This means that any correct participant's prior 2a is received by all correct participants. If there are no prior 2as, there will now be 2as with value $\mathrm{V}(x)$.

By the end of period 5, y will be delivered to all correct participants. They will then send 1bs to each other.

By the end of period 8, all 2as generated by correct participants with values $\neq \mathrm{V}(y)$ will be buried.

By the end of period 10, z is delivered to all correct participants. All correct participants must respond with valid 1bs.

By the end of period 11, all correct participants receive a quorum (there is a correct quorum, by the definition of Correct) of valid 1bs, and so produce 2as.

By the end of period 12, a quorum of 2as with ballot $\mathrm{b}(z)$ have been delivered. These form q_a. □

Introducing Artificial Delays in a Semi-Synchronous Network

A *Semi-Synchronous* network is one in which there exists some (possibly unknown) constant duration Δ such that all messages sent arrive within Δ. If the set of proposers is finite (it could be limited to the set of participants in any observer's quorum), then each can be assigned specific times they are allowed to propose. For example, if proposals include a timestamp, and all correct participants delay receipt of a 1a until after that time, there could be a limited set of timestamps each proposer is allowed to use.

These allowed times can be allocated in turns: each participant gets a span of time during which they can propose, and no one else can, and these turns are allocated in a round-robin fashion. If turns get exponentially longer with time (from some pre-determined start time), then for any finite maximum message delay Δ, and any finite maximum clock skew Δ', the network assumption will eventually be met during the turn of some correct participant, with a turn of duration $> 13(\Delta + \Delta')$. The correct participant need only propose once, wait a third of its turn, and then propose twice more, using the value of its highest known 2a.

4.7 Repeated Consensus

In maintaining, for instance, an ordered log, it is useful for observers to *decide* on the value which goes in each *slot*, traditionally starting at slot 0, then 1, etc. In general, one might want to prohibit filling a slot before a previous slot has been filled. With homogeneous observers, one might say that *1a* messages should be

ignored unless they can show that consensus for the previous slot was reached. For instance, a participant might ignore a *1a* for slot n unless it references *2a* messages signed by a quorum of participants which share a ballot and a value identified as belonging in slot $n-1$. In other words, a participant can demand *proof of consensus* for slot $n-1$ before filling slot n (for $n \neq 0$).

Heterogeneous observers makes this concept more difficult. What if the previous slot has been filled for one observer, but not for another? What if they are filled with different values for different observers? We describe a few possible solutions:

4.7.1 Allow slots to be filled in any order.

Each consensus protocol for each slot is fully independent. This is easier to implement, technically correct, and probably acceptable for some applications.

4.7.2 A *1a* for slot s is a *1a* for slot $s-1$.

Suppose each *1a* for slot $s > 0$ is required to reference a *1a* for slot $s-1$. Furthermore, each correct participant delays receipt of each *1a* until it has received and acted on the *1a*s referenced. Other than that, we treat all slots independently.

No slot could be filled for any observer without consensus at least having *begun* for all prior slots. This does not guarantee, however, that consensus has yet *finished* for all prior slots. However, given termination (§ 4.2.3), it *will finish*, and so all prior slots *will be filled*.

This does not guarantee that the value decided in slot s references the value decided in slot $s-1$.

4.7.3 Keep track of proof of consensus per observer.

A client could put all known proofs for slot $s-1$ in each *1a* for slot s, and participants only consider those observers in the *2a* phase for the ballot that *1a* begins. This pushes the duty of tracking proofs of consensus onto the client.

Later *1a* messages might feature more observers, thus enabling more observers to achieve consensus for slot s.

This is compatible with the *1a* for slot s references a *1a* for slot $s-1$ solution (§ 4.7.2).

4.8 Examples

In §4.5.1, we contrast Heterogeneous Consensus with traditional consensus in three ways:

- Heterogeneous failures
- Heterogeneous participants
- Heterogeneous observers

We illustrate the advantages of heterogeneity through examples with all eight combinations of Heterogeneous or Homogeneous failures, participants, and observers.

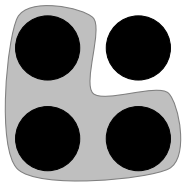

Figure 4.3: Fully Homogeneous Example: participants are shown as black circles, and a quorum is shown as a shaded region.

4.8.1 Fully Homogeneous

Consider a traditional byzantine-tolerant consensus protocol (such as the one described in § 4.4), with 4 participants, tolerating any 1 byzantine failure. Quorums (for any observer) consist of any 3 participants. We illustrate this traditional scenario in Fig. 4.3.

4.8.2 Heterogeneous Failures

Heterogeneous Consensus can express protocols wherein observers and participants are homogeneous, but mixed failures [134] are allowed. For instance, consider a protocol with 6 participants, tolerating at most 1 byzantine failure, and 1 additional crash failure. Quorums (for any observer) consist of any 4 participants. We illustrate this scenario in Fig. 4.4.

Note that in order to tolerate 2 total failures, a homogeneous byzantine-fault-tolerant consensus protocol would need at least 7 participants. Heterogeneity spares the expense (in latency and resources) of an unnecessary additional participant.

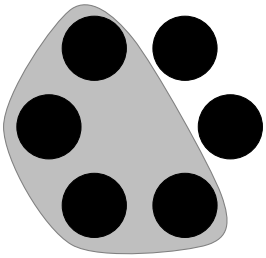

Figure 4.4: Heterogeneous Failures Example: participants are shown as circles, and a quorum is shown as a shaded region.

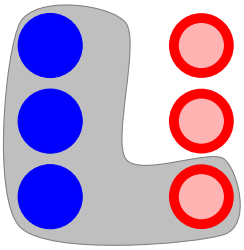

Figure 4.5: Heterogeneous Participants Example: Here, we draw one group of participants as solid blue circles, and the other as hollow red circles. A quorum is shown as a shaded region.

4.8.3 Heterogeneous Participants

Heterogeneous Consensus can express protocols wherein observers and failures are homogeneous, but not all participants are the same. For instance, consider a protocol with 6 participants, divided into two groups of 3. We tolerate up to 2 byzantine failures, but only so long as *all failures are in one group*. This makes the participants heterogeneous: it matters *which* two participants fail. Quorums (for any observer) consist of 3 participants from one group, and one participant from the other. This scenario is illustrated in Fig. 4.5.

Note that in order to tolerate 2 total failures, a homogeneous byzantine-fault-tolerant consensus protocol would need at least 7 participants. Heterogeneity spares the expense (in latency and resources) of an unnecessary additional partici-

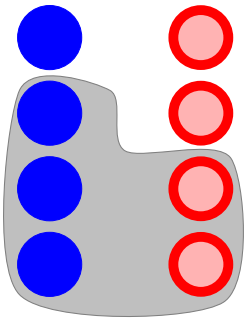

Figure 4.6: Heterogeneous Failures and Participants Example: Here, we draw one group of participants as solid blue circles, and the other as hollow red circles. A quorum is shown as a shaded region.

---

pant. It is possible to express this kind of heterogeneity with quorums in byzantine Paxos [85].

4.8.4 Heterogeneous Failures and Participants

Heterogeneous Consensus can express protocols wherein observers are homogeneous, but with mixed failures, heterogeneous participants. For instance, consider a protocol with 8 participants, divided into two groups of 4. We tolerate up to 2 byzantine failures, and 1 additional crash failure, but only so long as *all byzantine failures are in one group, and at most 2 failures occur in the same group.* This makes the participants heterogeneous: it matters *which* participants fail. Quorums (for any observer) consist of 3 participants from one group, and 2 participants from the other. We illustrate this example in Fig. 4.6.

Note that in order to tolerate 3 total failures, a homogeneous byzantine-fault-tolerant consensus protocol would need at least 10 participants. Heterogeneity spares the expense (in latency and resources) of 2 unnecessary additional participants.

To address a similar situation with homogeneous failures, we'd need still need 10 participants (to tolerate 2 byzantine failures in one group, and 1 in the other). To tolerate 2 byzantine failures and one crash failure (so heterogeneous failures) with homogeneous participants, we'd need 9 total participants. The additional detail of heterogeneous participants spares the expense (in latency and resources) of 2 unnecessary additional participants, as opposed to just heterogeneous failures.

4.8.5 Heterogeneous Observers

Heterogeneous Consensus can express protocols wherein observers have different failure assumptions, but each assumes participants and failures are homogeneous. This is the sort of scenario the original Ripple consensus protocol [123] tried to address, although it lacks liveness [34].

Participant Disagreement

We discussed a participant disagreement example in § 4.3.2, and we expand upon it here.

Suppose observers agree that they want to tolerate 1 byzantine failure out of 4 participants, but disagree about who the 4 relevant participants are. Suppose there are 4 observers: 2 red and 2 blue, as well as 5 participants: 1 red, 1 blue, and 3 black. The participants are illustrated in Fig. 4.7.

The red observers want to agree if there is at most 1 byzantine failure amongst the red and black participants, even if the blue participant has failed. Likewise, the blue observers want to agree if there is at most 1 byzantine failure amongst

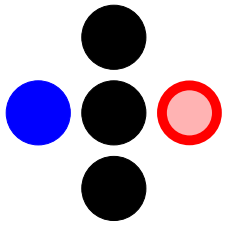

Figure 4.7: Membership Disagreement Example: Here, we draw one the blue participant as a solid blue circle, the red participant as a hollow red circle, and the black participants as black circles.

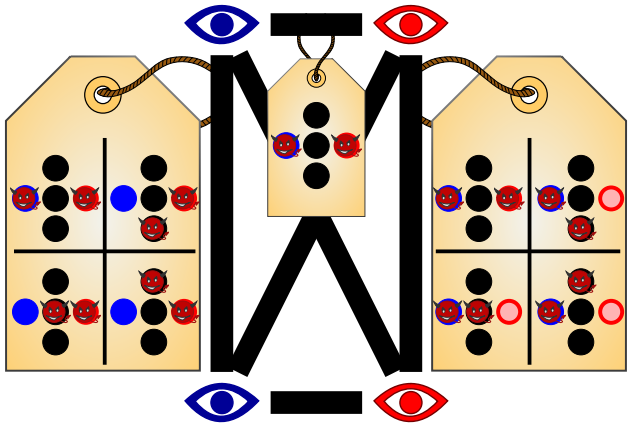

Figure 4.8: Membership Disagreement Observer Graph: Observers are drawn as eyes, with darker blue observers on the left, and lighter, outlined red observers on the right. Note that the edges are labeled with sets of different universes in which the pair of observers want to agree. In each universe shown, byzantine failures are marked with a demonic face. All the edges except the rightmost and leftmost share the same label, in the middle.

---

the blue and black participants, even if the red participant has failed. The red observers and blue observers acknowledge that they may disagree with each other if a black participant fails, but otherwise they want to agree. Thus we draw the observer graph (§ 4.3) in, Fig. 4.8.

For the red observers, quorums are any 3 red or black participants, while for the blue observers, quorums are any 3 blue or black participants. Quorums are shown in Fig. 4.9. Note that in order to tolerate 2 total failures, a homogeneous byzantine-

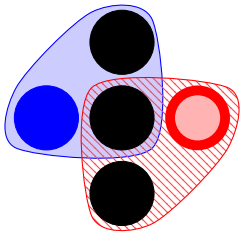

Figure 4.9: Membership Disagreement Quorums: A quorum for the blue observers is shown in the light solid blue region, and a quorum for the red observers is shown as a striped red region.

---

fault-tolerant consensus protocol would need at least 7 participants. Heterogeneity spares the expense of 2 unnecessary additional participants.

Failure Disagreement

Alternatively, observers might disagree about the types of failures they expect. Even if each observer expects homogeneous failures, observers' expectations may differ.

For example, consider a protocol with 5 participants. Two observers, called *blue*, want to agree so long as there isn't more than one failure, even if that failure is byzantine. Another two observers, called *red*, want to agree so long as there are no more than 2 crash failures, but accept that they may disagree if there is a byzantine failure. The red observers and the blue observers want to agree if there is no more than 1 crash failure, but accept that they may disagree if there is a byzantine failure or more than 1 crash failure. Thus we draw the observer graph (§ 4.3), in Fig. 4.10.

For the red observers, quorums are any 3 participants, while for the blue observers, quorums are any 4 participants. Quorums are illustrated in Fig. 4.11.

Figure 4.10: Failure Disagreement Observer Graph Observers are shown as eyes, with darker blue observers on the left, and lighter, outlined red observers on the right. Note that the edges between each pair of observers are labeled with the set of failures they want to tolerate. Crash (safe, but not live) failures are denoted with a skull, and byzantine failures with a devil. All edges except the rightmost and leftmost share a central label.

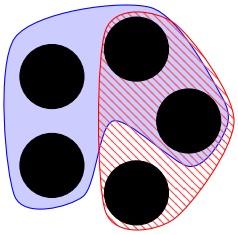

Figure 4.11: Failure Disagreement Example: A quorum for the blue observers is shown in the light solid blue region, and a quorum for the red observers is shown as a striped red region.

Note that in order to tolerate 2 total failures, a homogeneous byzantine-fault-tolerant consensus protocol would need at least 7 participants. Heterogeneity spares the expense (in latency and resources) of 2 unnecessary additional participants.

4.8.6 Heterogeneous Observers and Failures

Heterogeneous Consensus can express protocols wherein observers have different failure assumptions, and failures are heterogeneous, but participants are homoge-

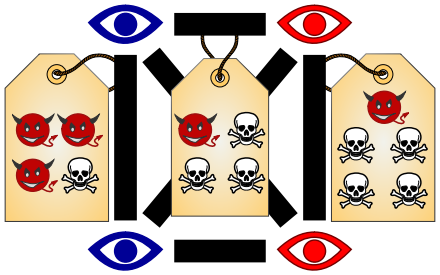

Figure 4.12: Heterogeneous Observers and Failures Observer Graph: Observers are drawn as eyes, with darker blue observers on the left, and lighter, outlined red observers on the right. Note that the edges between each pair of observers are labeled with the set of failures they want to tolerate. Crash failures are denoted with a skull, and byzantine failures with a devil. All edges except the rightmost and leftmost share a central label.

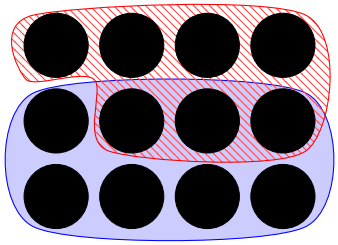

Figure 4.13: Heterogeneous Observers and Failures Example: A quorum for the blue observers is shown in the light solid blue region, and a quorum for the red observers is shown as a striped red region.

---

neous. Consider a protocol with 12 participants, and 4 observers. Two observers, called *blue*, want to agree so long as there isn't aren't more than 3 byzantine failures, and 1 additional crash failure. Another two observers, called *red*, want to agree so long as there are no more than 1 byzantine failure, and 4 additional crash failures. The red observers and the blue observers want to agree if there is no more than 1 byzantine and 3 additional crash failures. They accept that they may disagree otherwise. Thus we draw the observer graph (§ 4.3) in Fig. 4.12.

For the red observers, quorums are any 7 participants, while for the blue observers, quorums are any 8 participants. Example quorums are shown in Fig. 4.13.

Note that in order to tolerate 5 total failures, a homogeneous byzantine-fault-tolerant consensus protocol would need at least 16 participants. Heterogeneity spares the expense (in latency and resources) of 4 unnecessary additional participants.

To simultaneously tolerate all observers' worst fears, (3 byzantine and 2 additional crash failures), we'd need 14 participants. The additional detail of heterogeneous participants spares the expense (in latency and resources) of 2 unnecessary additional participants, as opposed to just heterogeneous failures.

4.8.7 Heterogeneous Observers and Participants

Heterogeneous Consensus can express protocols wherein observers have different failure assumptions, and participants are heterogeneous, but failures are homogeneous. Consider a protocol with 8 participants, and 4 observers. The participants are divided into two groups of 4: *blue* and *red*.

All observers want to agree whenever at most 1 participant is byzantine. Two observers, called *blue*, also want to agree with each other whenever at most 1 blue and 2 red participants are byzantine. Two observers, called *red*, also want to agree with each other whenever at most 1 red and 2 blue participants are byzantine.

Any 3 blue and 2 red participants form a quorum for the blue observers, and any 2 blue and 3 red participants form a quorum for the red observers. Example quorums are illustrated in Fig. 4.14.

In order to tolerate all the universes any observer believes possible, a consensus with homogeneous observers would need at least one more participant. Taking

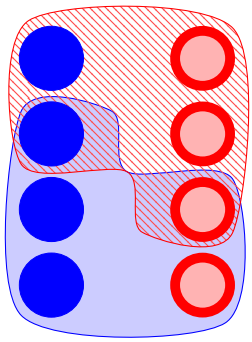

Figure 4.14: Heterogeneous Observers and Participants Example: Here, we draw one group as solid blue circles, and the other as hollow red circles. A quorum for the blue observers is shown in the light solid blue region, and a quorum for the red observers is shown as a striped red region.

Heterogeneous observers into account spares the expense of that unnecessary participant.

Since a single observer can tolerate 3 total failures, a protocol with homogeneous participants would require at least 10 participants. Heterogeneity spares the expense of 2 unnecessary additional participants.

4.8.8 Heterogeneous Observers, Failures and Participants

Heterogeneous Consensus can express protocols wherein observers have different failure assumptions, and failures are homogeneous, and so are participants. Consider a protocol with 9 participants, and 4 observers. The participants are divided into 3 groups of 3: *blue*, *black*, and *red*.

All observers want to agree when at most 1 blue, 1 black, and 1 red participant have crashed. 2 observers, called *blue*, want to agree with each other whenever at most 1 blue participant has crashed, 1 black participant is byzantine, and all

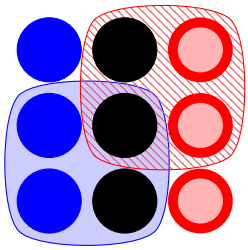

Figure 4.15: Heterogeneous Observers, Participants, and Failures Example: Here, we draw one group as solid blue circles, and the other as hollow red circles. A quorum for the blue observers is shown in the light solid blue region, and a quorum for the red observers is shown as a striped red region.

---

the red participants are byzantine. 2 observers, called *red*, want to agree with each other whenever at most 1 red participant has crashed, 1 black participant is byzantine, and all the blue participants are byzantine.

Any 2 blue and 2 black participants form a quorum for the blue observers, and any 2 red and 2 black participants form a quorum for the red observers. Example quorums are illustrated in § 4.8.8.

For a fully homogeneous consensus to tolerate 4 byzantine failures would require 13 participants, so heterogeneity spares the cost of 4 additional unnecessary participants.

4.9 Implementation

4.9.1 Charlotte

Since Heterogeneous Consensus is designed for cross-domain applications where different parties have different trust assumptions, it is well-suited for blockchain appli-

cations. With the Charlotte framework (chapter 3), we built Fern Servers (§3.4.2) that acted as participants, and a subtype of integrity attestation (§3.3.8) featuring decision sets (Def. 19), as a "proof of consensus" for a given observer.

4.9.2 Meet

Recall that a *meet* Integrity Attestation is a subtype of two other types of Integrity Attestation (§3.6.3). If an attestation of type α proves a block belongs in one data structure, and an attestation of type β proves a block belongs in another data structure, then an attestation of type $\alpha \sqcap \beta$ proves the block belongs in both. Not all types of integrity attestation have a natural meet.

Heterogeneous Consensus, however, provides an expressive language for observers to specify meet types: the observer graph (§4.3). If one group of observers care about one data structure, and another group cares about another, then when they want to commit a block to both, they need to specify the edges between the two groups: the conditions under which they want the commit to be atomic.

To preserve maximum *safety* (but not maximum liveness), the quorum necessary for an protocol with the meet type is the union of one quorum from each component type. In other words, to make an observer decide with an integrity attestation of type $\tau_r \sqcap \tau_b$, you need all the participants it would take to make an attestation of type τ_r *and* all the participants it would take to make an attestation of type τ_b. With this construction, we can atomically commit a single block onto multiple Heterogeneous Consensus chains.

To preserve maximum *liveness*, observers should simply not demand to agree between the two groups. Each will commit blocks independently, and they will

likely not agree (it's not very safe). With this construction, Heterogeneous Consensus neatly describes independent consensus protocols as a single protocol.

4.9.3 Charlotte Representation

We implemented a prototype of Heterogeneous Consensus as a Fern service. Integrity attestations are specific to each observer's assumptions. We use Charlotte blocks as messages in the consensus protocol itself, so attestations can reference messages demonstrating that consensus was achieved.

In our implementation, there is a separate *light client* that does not participate in the consensus: it merely request an integrity attestation from a Fern server, which acts as the client in the consensus protocol. Including receiving a request from and sending an attestation to the light client, the process has a minimum latency of 5 messages.

In our implementation, quorums representing trust configurations are encoded as blocks. Each Heterogeneous Consensus blockchain includes a reference to such a block in its root, ensuring everyone agrees on the configuration. To append a block to the chain, a client requests an integrity attestation for some observer, specifying proposed block and height. To propose one block be appended to multiple chains, a client can request an integrity attestation that is the maximum safety meet (§3.6.3) of the integrity attestations needed for both chains. The Fern servers then run a round of consensus in which each quorum includes one quorum of the consensus necessary for each chain.

For the purposes of demonstrating the Charlotte framework, our experiments with Heterogeneous Consensus are *symmetric*: all observers want to agree under

the same conditions. For instance, observers might trust 4 Fern servers to maintain a chain, expecting no more than one of them to be byzantine. This is the only way one can compare Heterogeneous Consensus with existing implementations.

4.9.4 Evaluation

In order to evaluate the feasibility of consensus-based blockchains, and multi-chain blocks in Charlotte, we built several chains with Heterogeneous Consensus, and ran 5 types of experiments. With our artificial network latency of 100ms per message send, the theoretical lower bound on consensus latency is 500 ms, and maximum throughput per chain is 2 blocks/second. Each experiment recorded the latency light clients experience in appending their own blocks to the chain, as well as system-wide throughput. All experiments used single-core VMs with 8 GB RAM, except as noted.

Single Chain

In these experiments, a client appends 2000 successive blocks to one chain. Mean latency is 527 ms for a chain with 4 Fern servers and $538ms$ for 7 Fern servers. Since the best possible latency is 500 ms, these results are promising. Overheads include cryptographic signatures, verification, and garbage collection.

Parallel

As the darker green lines in Fig. 4.16 show, independent Heterogeneous Consensus chains have independent performance. In these experiments, we simultaneously

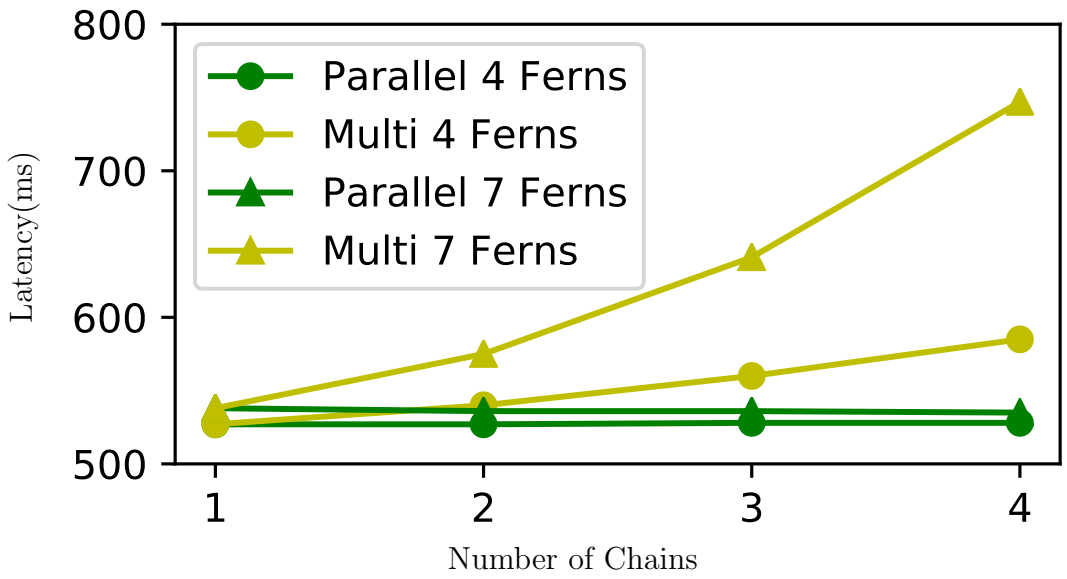

Figure 4.16: Heterogeneous Consensus Multichain and Parallel experiments. In Parallel experiments, each chain operates independently (and has its own client). In Multichain experiments, one client tries to append all blocks to all chains. Optimal latency is 500 ms.

---

ran 1–4 independent chains, each with 4 or 7 Fern servers. In each experiment, a client appends 2000 successive blocks to one chain. There is no noticeable latency difference between a single chain and many chains running together. Throughput scales with the number of chains (and inversely to latency). This scalability is the fundamental advantage of a blockweb over forcing everything onto one central blockchain.

Multichain shared blocks

Shared (joint) blocks facilitate inter-chain interaction (§ 3.6.3). In these experiments, a single client appends 1000 shared blocks to 2–4 chains, each with 4 or 7 Fern servers. As the yellow lines in Fig. 4.16 show, latency scales roughly linearly with the number of chains.

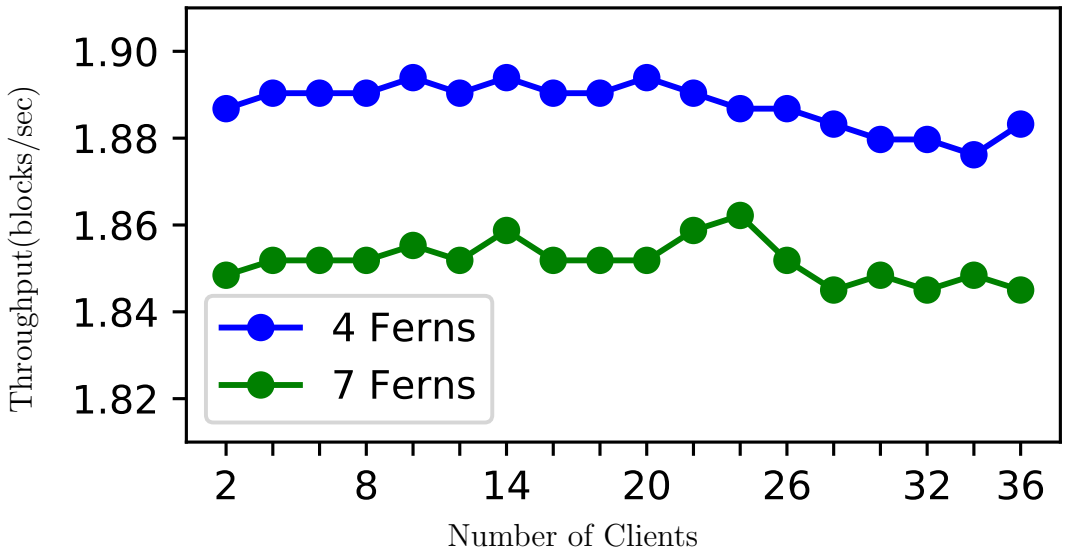

Figure 4.17: Throughput of Heterogeneous Consensus under contention. 2–36 clients try to append 2000 blocks to just one chain. Optimal throughput is 2 blocks/sec.

Contention

In these experiments, all clients simultaneously contend to append 2000 unique blocks to the same chain. We measured the blocks that were actually accepted into slots 500–1500 of the chain. We used 2–36 clients, and chains with 4 or 7 Fern servers, configured with 2 GB RAM. Like Byzantized Paxos [85], Heterogeneous Consensus can get stuck under contention and occasionally requires a dynamic timeout to automatically trigger a new round. Chain throughput is shown in Fig. 4.17. Our chains, on average, achieved 1.88 blocks/sec throughput for 4 Fern servers and 1.85 blocks/sec for 7, not far from the 2 blocks/sec optimum. Throughput does not decrease much with the number of clients.

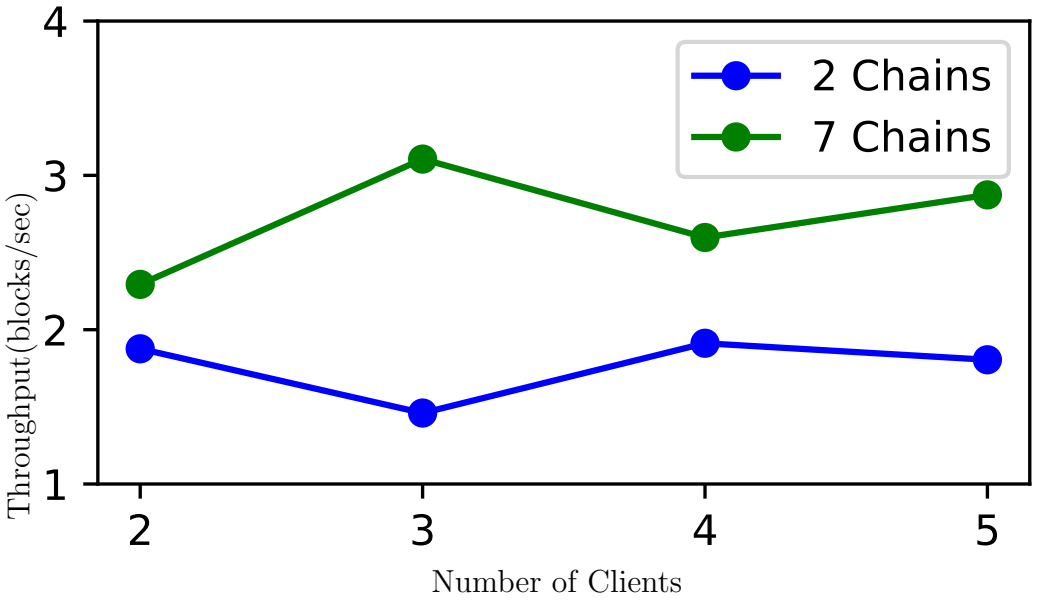

Figure 4.18: Throughput of Heterogeneous Consensus mixed-workload experiment (4 Fern servers).

4.9.5 Mixed

These experiments attempt to simulate a more realistic scenario by including all 3 types of workload. 2–5 clients contend to append blocks onto either 2 or 7 chains, each with 4 Fern servers. On each block, a client tries to append a shared block to two random chains with probability 10% and otherwise tries appending to a random single chain. The results are in Fig. 4.18. Throughput can be over 2.0 blocks/sec because multiple clients can append blocks to different chains in parallel. Mean throughput is 1.8 blocks/sec and 2.7 blocks/sec for 2 and 7 chains respectively, which is expected because the 2-chain configuration has more contention.

Heterogeneous Consensus scales horizontally with multiple chains running in parallel. Furthermore, throughput does not decrease much with more clients involved.

This gives us ability to make progress even with lots of clients connecting to the same chain concurrently. We also notice that, the number of Ferns servers play major roles for the latency performance. With a small group of Fern servers, Heterogeneous Consensus can almost reach 500 ms, which is the best we can get. Although the latency increases linearly with respect to the number of Ferns, for some applications, it is possible to break down big shared blocks into a set of small shared blocks. For example, in § 3.6.5, we discuss how to break a k-chain transaction into a $\log k$-depth graph whose nodes are small two-chain transactions. By following the same strategy, the latency would be reduced to $t \times \log k$, where t is the average latency for completing a 2-chain block. Since our Heterogeneous Consensus implementation is just a prototype, we believe that with further efforts in optimization, average latency performance can be improved.

4.10 Future Work

We have generalized what it means to have Non-Triviality, Integrity, Agreement and Termination in a heterogeneous setting, and designed and implemented a consensus algorithm that meets these properties with minimum theoretical latency. However, Heterogeneous Consensus is far from perfect. Here are a few ways it can almost certainly improved, without altering its core concepts.

4.10.1 Network Assumption and Termination

While Paxos and PBFT are semi-synchronous consensus protocols [83, 30], recent advances in cryptography have made fully asynchronous, probabilistically termi-

nating consensus protocols viable [103, 3]. These have the advantage of never needing to insert artificial delays in order to guarantee termination. It is possible that Heterogeneous Consensus could be adapted to this setting, using a similar "shared coin flip" mechanism, however it may require more interesting cryptography to deal with non-uniform quorums.

Even in the semi-synchronous setting, there are almost certainly better timing strategies than those we present (§ 4.6.7). Optimizations like the leader/view change used in pbft [30] or the batching used in BFT-SMart [21] could greatly improve performance.

4.10.2 Bandwidth

If each message bounces off of each participant, and each message is sent to each participant, and each phase features each participant sending a message, then the communication overhead is $O(n^3)$ messages. While the Charlotte architecture allows each message to avoid copying the proposed value, that can still be a lot of overhead. A fairly naive optimization would be to allow participants not to bounce all messages they receive, and instead request them only if the original sender didn't broadcast the message fast enough. This could reduce communication overhead to $O(n^2)$, in the absence of Byzantine failures.

Using advanced cryptographic techniques, protocols like Hot-Stuff achieve consensus in $Q(n)$ bandwidth overhead. It may be possible to adapt Heterogeneous Consensus with some of these optimizations.

4.10.3 Programming

Implementations like BFT-SMart [21] have put a great deal of effort into optimizing their implementations, discovering better data structures and communication protocols along the way. Our implementation is for from optimal. Better memory layout, data structures, and even parallelism and concurrency control techniques are certainly possible.

4.11 Related Work

Heterogeneous Consensus is based on Leslie Lamport's byzantine-fault-tolerant variant [85] of Paxos [83, 84]. Paxos is usually implemented as heterogeneous in all three ways (§ 4.5.1). However, byzantine Paxos, as Lamport defined it, is heterogeneous in at least 1 way:

- Heterogeneous *participants*: Paxos uses quorums. Not all participants need be of equal worth, but in some sense, all quorums are.

Furthermore, as we've defined byzantine Paxos (§ 4.4), it is heterogeneous in another way:

- Heterogeneous *failures*: byzantine paxos can have *mixed failures* [134] so long as our assumptions about quorum intersections featuring a safe participant and at least one quorum being live are met.

Ripple's Cobalt protocol [94] (which addresses issues in Ripple's earlier consensus protocol [123]) heterogeneous observers:

- Heterogeneous *observers*: Each observer specifies a set of participants they partially trust, and it only works if those sets intersect enough.

Cobalt has homogeneous failures, and within "essential subsets" [94], participants are homogeneous as well.

Stellar Consensus [96] is heterogeneous in two ways:

- Heterogeneous *observers*: Each observer specifies their "slices" [96], which are sets of participants they partially trust.
- Heterogeneous *participants*: In a sense, each slice for each observer is equivalent, but not participants individually.

Unlike Stellar, Heterogeneous Consensus has mixed (heterogeneous) failures. Furthermore, we are unaware of any consensus protocol with heterogeneous observers that matches Heterogeneous Consensus' best-case latency. Heterogeneous Consensus inherits byzantine Paxos' 3-message-send best case latency, which is optimal for a consensus tolerating $\lceil \frac{n}{3} \rceil - 1$ failures in the homogeneous byzantine case or $\lceil \frac{n}{2} \rceil - 1$ failures in the homogeneous crash case.

We are not aware of any other consensus with heterogeneous observers and mixed failures.

4.12 Discussion

Heterogeneous Consensus is the first consensus algorithm with heterogeneous participants, failures, and observers. It facilitates a more nuanced approach that can

save time and resources, or even make previously unachievable consensus possible. This approach is well-suited to federated systems across heterogeneous trust domains, such as blockchains. We use Charlotte to demonstrate working, composable blockchains with well-defined integrity and availability properties, and minimal overhead, using Heterogeneous Consensus. Heterogeneous Consensus is a useful building block for systems that seek to embrace least ordering, fault tolerance, and heterogeneity.

CHAPTER 5

CONCLUSIONS

In this dissertation, I have discussed building distributed systems with serializable transactions. To address the shortcomings of existing distributed systems, I embrace least ordering, fault tolerance, and heterogeneity in three related projects: Safe Serializable Secure Scheduling, Charlotte, and Heterogeneous Consensus. Together, these open new doors in system design for increased performance, flexibility, and security.

5.1 Safe Serializable Secure Scheduling

Fabric [90] provides serializable transactions among heterogeneous participants. However, existing scheduling protocols, while trying to preserve Least Ordering, make heterogeneous assumptions. I prove that they can leak information when different participants are trusted with different data, and in fact execute an attack. I then go on to develop a novel category of *relaxed monotonic* transactions which can be scheduled safely, and a protocol, *staged commit*, to schedule them. We integrate this into Fabric's compiler and runtime, and show that it imposes minimal, although measurable, overhead.

This serves not only as a useful protocol for future systems, but as an example of how trickly serialization can be, especially when heterogeneity is involved.

5.2 Charlotte

Modern blockchain systems have taken a brute-force "order everything, tolerate as many failures as possible" approach to transactions. This carries multiple shortcomings:

- Serializing all transactions, as opposed to just maintaining serializ*ability*, abandons the Least Ordering principle: as a result they are tremendously expensive and slow.
- In order to make applications compositional, blockchains force them all to use the same chain. This ignores the heterogeneity of observers for different applications, and instead tries to tolerate the sum of all their fears, which is needlessly expensive.

In chapter 3, a framework for Composable Authenticated Distributed Data Structures, I address these shortcomings while preserving prized properties like self-authenticating data, and provable commits. With Charlotte, data structures can reference each other, and even share blocks, while retaining verifiable formal properties. I demonstrate how existing ADDSs can be replicated within the Charlotte framework, including Git, Bitcoin, and Timestamping, with minimal overhead. I also examine how, with Least Ordering, the actual transactions in the Bitcoin payment history could be committed about 70 times faster.

5.3 Heterogeneous Consensus

Unfortunately, Staged Commit loses liveness when participants crash. Charlotte is built to allow a variety of integrity mechanisms to commit blocks (which can represent transactions) to data structures.

To fully embrace heterogeneity, in chapter 4, I design Heterogeneous Consensus, the first consensus algorithm that can be tailored for heterogeneous participants, observers, and failures. I used the Charlotte framework to construct Heterogeneous Consensus servers, which in turn plug in as an integrity mechanism for Charlotte data structures. I created a rich model in which to express heterogeneous intergrity properties: the Observer Graph. This in turn can be used to facilitate multi-structure fault tolerant transaction commits using Heterogeneous Consensus and Charlotte, embracing Least Ordering, Heterogeneity, and Failure Tolerance.

5.4 Future Work

Each of these projects have their own directions to expand in the future.

5.4.1 Safe Serializable Secure Scheduling

I don't yet know if relaxed monotonicity is the largest set of transactions that can be securely scheduled. There is room between our necessary condition, Thm. 3, and our sufficient condition, relaxed monotonicity, to discover precisely what is necessary and sufficient for transactions to be securely serializable. There is also room for new, possibly more efficient transaction ordering algorithms.

Our analysis also does not explicitly take availability labels into account. It remains to be seen if this carries some hidden difficulties.

5.4.2 Charlotte

Charlotte formalizes integrity and availability properties, but what about confidentiality properties? Can references specify the secrecy of a block's existence, and complete the "CIA Triad"? For example, we might imagine blocks carrying statements of the form "references to this block must be at least this encrypted," and all references carrying proofs that they are appropriately encrypted. Advanced cryptography may be necessary to check these proofs without decrypting anything, if that is desirable.

There are hosts of Charlotte applications that remain unrealized. One avenue that would help create them is to create a new programming model, and a language to go with it, for applications built on ADDSs. For example, we can envision objects (as in Object oriented language) as the unit of serializability, and maintain a chain of transactions for each object. Each transaction, represented as a block, must be in all the chains of all the objects it touches. This language would need availability and integrity labels on data, and could enforce the same style of security properties that Fabric does with Information Flow Control.

5.4.3 Heterogeneous Consensus

Heterogeneous consensus can be improved with a myriad of optimizations for speed, bandwidth, and improved termination guarantees, but these are best left to their own section (§ 4.10).

Like 2 Phase Commit, Staged Commit is not live if a participant fails. While Heterogeneous Consensus can order transactions in the presence of failures, it is designed only around integrity and availability, and does not consider confidentiality. Someday, the world may need a new scheduling protocol that considers all three, and tolerates failures.

In addition to dealing with Confidentiality, Heterogeneous Consensus could be adapted to take into account more types of failures, including rational agents [5] and fail-stop [120]. This would require altering the nature of the Observer Graph in order to detail even more possible failure scenarios.

5.4.4 Final Thoughts

Bringing Least Ordering, Fault Tolerance, and Heterogeneity to serialized transactions remains a rich research area beyond these projects. New systems, languages, and programming paradigms can gain efficiency and security by embracing these ideals, and in doing so open the doors to hosts of new applications.

APPENDIX A

CHARLOTTE APPENDICES

A.1 Bitcoin Transactions in Two Accounts or Fewer

In Bitcoin, it is advantageous to combine many small transfers of money into big ones, with many inputs and many outputs. This improves anonymity and performance. In the real financial system of the USA, however, all monetary transfers are from one account to another. They are all exactly two chain transactions.

We can simulate this limitation by refactoring each Bitcoin UTXO as 2 UTXOs, and each Bitcoin transaction as a DAG of transactions with depth:

$$\lceil log_2(max(\text{number of inputs, number of outputs})) \rceil$$

To do this, we create

$$n \triangleq 2^d$$

chains, each of which is

$$d \triangleq \lceil log_2(max(\text{number of inputs, number of outputs})) \rceil$$

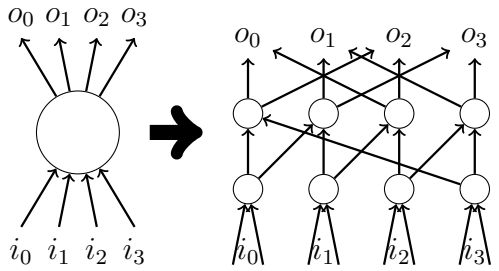

Figure A.1: Converting 4 inputs and 4 outputs to a graph of 2-account transactions.

long. We call these chains C^0 through C^n. Original input UTXO i corresponds to both inputs to the first transaction of chain i. Original output UTXO j corresponds to one output of each of the last transactions from chains j and $\left(j + 2^{d-1}\right) \bmod n$. For $0 \leq k < (d-1)$, the outputs of the k^{th} transaction in chain i, called C_k^i, go to C_{k+1}^i, and:

$$C_{k+1}^{\left(i+2^j\right) \bmod\ n}$$

The outputs of C_d^i go to the UTXOs corresponding with output i, and output $\left(i + 2^{d-1}\right) \bmod n$. Each transaction divides its output values proportionately to the sums of the final output values reachable from each of the transaction's outputs. Fig. A.1 is an example transformation from a 4-input, 4-output transaction to a DAG of depth 2 using all 2-input, 2-output transactions.

BIBLIOGRAPHY

[1] Distributed transactions: .NET framework 4.6. https://msdn.microsoft.com/en-us/library/ms254973%28v=vs.110%29.aspx. Accessed: 2015-11-13.

[2] XA standard. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 3571–3571. Springer US, 2009.

[3] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message BFT devil. *CoRR*, abs/1803.05069, 2018.

[4] Marcos K. Aguilera, Joshua B. Leners, and Michael Walfish. Yesquel: Scalable sql storage for web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 245–262, New York, NY, USA, 2015. ACM.

[5] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 45–58, New York, NY, USA, 2005. ACM.

[6] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *Conference on Innovative Data Systems Research (CIDR)*, January 2011.

[7] Amazon. Cloud storage with AWS. https://aws.amazon.com/products/storage.

[8] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, November 1999.

[9] Owen Arden, Jed Liu, Tom Magrino, and Andrew C. Myers. Fabric 0.3. Software release, http://www.cs.cornell.edu/projects/fabric, June 2016.

[10] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *17th ACM Conf. on Computer and Communications Security (CCS)*, pages 297–307, October 2010.

[11] V. Atluri, S. Jajodia, and B. George. *Multilevel Secure Transaction Processing*. Advances in Database Systems. Springer US, 2000.

[12] Vijayalakshmi Atluri, Sushil Jajodia, Thomas F Keefe, Catherine D McCollum, and Ravi Mukkamala. Multilevel secure transaction processing: Status and prospects. *DBSec*, 8(1):79–98, 1996.

[13] Hillel Avni, Eliezer Levy, and Avi Mendelson. Hardware transactions in nonvolatile memory. In Yoram Moses, editor, *Distributed Computing*, volume 9363 of *Lecture Notes in Computer Science*, pages 617–630. Springer Berlin Heidelberg, 2015.

[14] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013.

[15] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electron. Notes Theor. Comput. Sci.*, 153(2):33–55, May 2006.

[16] Juan Benet. IPFS – content addressed, versioned, P2P file system. `https://ipfs.io`.

[17] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, May 1995.

[18] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[19] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[20] Elisa Bertino, Barbara Catania, and Elena Ferrari. A nested transaction model for multilevel secure database management systems. *ACM Trans. Inf. Syst. Secur.*, 4(4):321–370, November 2001.

[21] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 355–362. IEEE, 2014.

[22] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA,

April 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[23] Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 43–54, New York, NY, USA, 2009. ACM.

[24] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 49–60, Berkeley, CA, USA, 2013. USENIX Association.

[25] Vitalik Buterin, Danny Ryan, Hsiao-Wei Wang, Terence Tsao, and Chih Cheng Liang. Ethereum 2.0 phase 1 – shard data chains. `https://https://github.com/ethereum/eth2.0-specs/blob/master/specs/core/1_shard-data-chains.md`.

[26] C. Cachin and M. Vukolić. Blockchain Consensus Protocols in the Wild. *ArXiv e-prints*, July 2017.

[27] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *23rd ACM Symp. on Operating System Principles (SOSP)*, pages 143–157. ACM, 2011.

[28] Jon Callas, Lutz Donnerhacke, Hal Finney, David Shaw, and Rodney Thayer. OpenPGP message format. RFC 4880, RFC Editor, November 2007. `http://www.rfc-editor.org/rfc/rfc4880.txt`.

[29] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

[30] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. on Computer Systems*, 20:2002, 2002.

[31] Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Transaction chopping for parallel snapshot isolation. In Yoram Moses, editor, *Distributed*

Computing, volume 9363 of *Lecture Notes in Computer Science*, pages 388–404. Springer Berlin Heidelberg, 2015.

[32] Scott Chacon and Ben Straub. 7.11 git tools - submodules. In *Pro Git*, pages 299—318. Apress, 2 edition, 2014.

[33] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In *11th ACM Symp. on Principles of Distributed Computing*, PODC '92, pages 147–158, August 1992.

[34] Brad Chase and Ethan MacBrough. Analysis of the XRP ledger consensus protocol. *CoRR*, abs/1802.07242, 2018.

[35] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *IEEE Computer Security Foundations Symp. (CSF)*, pages 51–65, June 2008.

[36] Ariel Cohen, Ron van der Meyden, and Lenore D. Zuck. Access control and information flow in transactional memory. In *Formal Aspects in Security and Trust, 5th International Workshop, FAST 2008, Malaga, Spain, October 9-10, 2008, Revised Selected Papers*, pages 316–330, 2008.

[37] Bram Cohen. The BitTorrent protocol specification. `http://bittorrent.org/beps/bep_0003.html`, 2017.

[38] Wikipedia contributors. Magnet URI scheme — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Magnet%20URI%20scheme&oldid=888151611`, 2019. [Online; accessed 2019-03-25].

[39] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[40] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, pages 106–125, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[41] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. TARDiS: A branch-and-merge approach to weak consis-

tency. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1615–1628, June 2016.

[42] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *18th ACM Symp. on Operating System Principles (SOSP)*, pages 202–215, 2001.

[43] George Danezis and Sarah Meiklejohn. Centrally banked cryptocurrencies. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[44] Alex de Vries. Bitcoin energy consumption index. Technical report, Digiconomist, 2018.

[45] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key–value store. In *21st ACM Symp. on Operating System Principles (SOSP)*, 2007.

[46] Dorothy E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, 1976.

[47] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[48] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[49] Alexandros G. Dimakis, Kannan Ramchandran, Yunnan Wu, and Changho Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99:476–489, 2011.

[50] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *25th ACM Symp. on Operating System Principles (SOSP)*, pages 54–70, New York, NY, USA, 2015. ACM.

[51] Dominic Duggan and Ye Wu. Transactional correctness for secure nested transactions - (extended abstract). In *Trustworthy Global Computing - 6th*

International Symposium, TGC 2011, Aachen, Germany, June 9-10, 2011. Revised Selected Papers, pages 179–196, 2011.

[52] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Comm. of the ACM*, 19(11):624–633, November 1976. Also published as IBM RJ1487, December 1974.

[53] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. *Commun. ACM*, 61(7):95–102, June 2018.

[54] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985. Also published as MIT Laboratory of Science Technical Report MIT/LCS/TR-282, Cambridge, MA, 1982.

[55] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[56] Ethereum Foundation. Ethereum white paper. Technical report, Ethereum Foundation, 2018.

[57] Ethan Frey and Christopher Goes. Cosmos Inter-Blockchain Communication (IBC) Protocol. `https://cosmos.network`, October 2018.

[58] J. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. In *11th Workshop on Distributed Algorithms (WDAG)*, Saarbrücken, Germany, September 1997.

[59] JuanA. Garay and KennethJ. Perry. A continuum of failure models for distributed computing. In Adrian Segall and Shmuel Zaks, editors, *Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 153–165. Springer Berlin Heidelberg, 1992.

[60] Git large file storage. `https://git-lfs.github.com`, 2018.

[61] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[62] Google. Google cloud storage. `https://cloud.google.com/storage`.

[63] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Language Specification*. Oracle America, se 11 edition, Aug 2018.

[64] grpc: A high performance, open-source universal RPC framework. `https://grpc.io`, 2018.

[65] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[66] Sascha Hanse, Luca Favetella, Hans Svensson, Emin Mahrt, Tobias Lindahl, and Erik Stenman. æternity protocol. `https://aeternity.com`, January 2019.

[67] Andre Heinecke. How to setup an OpenLDAP-based PGP keyserver. `https://wiki.gnupg.org/LDAPKeyserver`, Dec 2017.

[68] Shidokht Hejazi-Sepehr, Ross Kitsis, and Ali Sharif. Transwarp-Conduit: Interoperable blockchain application framework. `https://aion.network/developers`, January 2019.

[69] Marc Horowitz. A PGP public key server. Technical report, MIT, Nov 1996.

[70] R. Housley, T. Polk, W. Ford, and D. Solo. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. Internet RFC-3280, April 2002.

[71] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC*, 2010.

[72] IEEE. *The Open Group Base Specifications Issue 7*. IEEE and The Open Group, 2018 edition, 2018.

[73] IOTA. Qubic: Quorum based computations powered by IOTA. `https://qubic.iota.org`.

[74] IPLD. `https://ipld.io`.

[75] Ari Juels and Burton S. Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM Conference on Computer and Com-*

munications Security, CCS '07, pages 584–597, New York, NY, USA, 2007. ACM.

[76] I. E. Kang and T. F. Keefe. Transaction management for multilevel secure replicated databases. *J. Comput. Secur.*, 3(2-3):115–145, March 1995.

[77] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. *2018 IEEE Symposium on Security and Privacy (Oakland)*, pages 583–598, 2018.

[78] B. Köpf and M. Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *2009 IEEE Computer Security Foundations*, July 2009.

[79] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, April 2013.

[80] Bogdan Kulynych, Wouter Lueks, Marios Isaakidis, George Danezis, and Carmela Troncoso. ClaimChain: Improving the security and privacy of in-band key distribution for messaging. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*, WPES'18, pages 86–103, New York, NY, USA, 2018. ACM.

[81] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.

[82] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. on Programming Languages and Systems*, 4(3):382–401, July 1982.

[83] Leslie Lamport. The Part-time Parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, May 1998.

[84] Leslie Lamport. Paxos made simple. pages 51–58, December 2001.

[85] Leslie Lamport. Byzantizing Paxos by refinement. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, pages 211–224, Berlin, Heidelberg, 2011. Springer-Verlag.

[86] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Trans. on Programming Languages and Systems*, 4(3):382–401, July 1982.

[87] B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, Palo Alto, CA, 1979.

[88] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 71–86, New York, NY, USA, 2015. ACM.

[89] Colin LeMahieu. Nano: A feeless distributed cryptocurrency network. Technical report, Nano, 2018.

[90] Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. Fabric: Building open distributed systems securely by construction. *J. Computer Security*, 25(4–5):319–321, May 2017.

[91] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, October 2009.

[92] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *23rd ACM Symp. on Operating System Principles (SOSP)*, 2011.

[93] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

[94] Ethan MacBrough. Cobalt: BFT governance in open networks. *CoRR*, abs/1802.07240, 2018.

[95] Chip Martel, Glen Nuckolls, Michael Gertz, Prem Devanbu, April Kwong, and Stuart G. Stubblebine. A general model for authentic data publication. `https://www.semanticscholar.org/paper/`

A-General-Model-for-Authentic-Data-Publication-Martel-Nuckolls/0491749101fea10b612d07cd774b6f960c3ef8ce, 2001.

[96] David Mazières. The Stellar consensus protocol: A federated model for internet-level consensus. https://www.stellar.org, April 2015.

[97] Trent McConaghy, Rodolphe Marques, and Andreas Müller. BigchainDB: a scalable blockchain database. https://www.bigchaindb.com/whitepaper, 2016.

[98] Daryl McCullough. Noninterference and the composability of security properties. In *IEEE Symp. on Security and Privacy*, pages 177–186. IEEE Press, May 1988.

[99] Ralph C. Merkle. Protocols for public key cryptosystems. In *IEEE Symp. on Security and Privacy*, page 122, Los Alamitos, CA, USA, 1980. IEEE Computer Society.

[100] Microsoft. Azure storage. https://azure.microsoft.com/services/storage.

[101] Sun Microsystems. JavaBeans (version 1.0.1-a). http://java.sun.com/products/javabeans/docs/spec.html, August 1997.

[102] Matthew P. Milano and Andrew C. Myers. MixT: A language for mixing consistency in geodistributed transactions. In *39th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2018. To appear.

[103] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 31–42, New York, NY, USA, 2016. ACM.

[104] Multiformats. https://multiformats.io.

[105] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[106] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1(2012):28, 2008.

[107] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[108] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[109] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. `https://plasma.io`, August 2017.

[110] Serguei Popov. The Tangle. Technical report, Iota, 2018.

[111] Dan R. K. Ports and Kevin Grittner. Serializable snapshot isolation in PostgreSQL. *Proc. VLDB Endow.*, 5(12):1850–1861, August 2012.

[112] Bart Preneel. Collision resistance. In Henk C. A. van Tilbor and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security*, pages 221–222, Boston, MA, 2011. Springer US.

[113] Protocol buffers. `https://developers.google.com/protocol-buffers/`, 2018.

[114] Yoav Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *18th Very Large Data Bases Conference (VLDB)*, August 1992.

[115] Team Rocket. Snowflake to Avalanche: A novel metastable consensus protocol family for cryptocurrencies. `https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5o2FYopNPVYwrRVGV`, May 2018.

[116] A. W. Roscoe. CSP and determinism in security modelling. In *IEEE Symp. on Security and Privacy*, pages 114–127, May 1995.

[117] Sheldon M Ross. *MySQL 8.0 Reference Manual*. Oracle, 2019.

[118] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[119] Andrei Sabelfeld and Andrew C. Myers. A model for delimited release. In *2003 International Symposium on Software Security*, number 3233 in Lecture Notes in Computer Science, pages 174–191. Springer-Verlag, 2004.

[120] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. on Computer Systems*, 1(3):222–238, August 1983.

[121] F. B. Schneider. The state machine approach: A Tutorial. Technical Report TR 86-600, Cornell University, Dept. of Computer Science, Ithaca, N. Y., December 1986.

[122] David A. Schultz and Barbara Liskov. IFDB: decentralized information flow control for databases. In *EUROSYS*, 2013.

[123] David Schwartz, Noah Youngs, and Arthur Britto. The Ripple protocol consensus algorithm. Technical report, Ripple Labs Inc, 2014.

[124] Michael L. Scott. Sequential specification of transactional memory semantics. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[125] Ravi Sethi. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM*, 29(2):394–403, 1982.

[126] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88, 2011.

[127] David Shaw. The OpenPGP HTTP Keyserver Protocol (HKP). Internet-Draft draft-shaw-openpgp-hkp-00.txt, IETF Secretariat, March 2003.

[128] Isaac Sheff, Tom Magrino, Jed Liu, Andrew C. Myers, and Robbert Van Renesse. Safe serializable secure scheduling: Transactions and the trade-off between security and consistency. In *23rd ACM Conf. on Computer and Communications Security (CCS)*, pages 229–241, October 2016.

[129] Isaac Sheff, Tom Magrino, Jed Liu, Andrew C. Myers, and Robbert van Renesse. Safe serializable secure scheduling: Transactions and the trade-off between security and consistency. Technical Report 1813–44581, Cornell University Computing and Information Science, August 2016.

[130] Isaac C. Sheff, Robbert van Renesse, and Andrew C. Myers. Distributed protocols and heterogeneous trust: Technical report. Technical Report arXiv:1412.3136, Cornell University Computer and Information Science, December 2014.

[131] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 325–336, New York, NY, USA, 2013. ACM.

[132] Abraham Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Concepts*. Windows XP update. Wiley, 2003.

[133] Emin Gün Sirer, Willem De Bruijin, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *11^{th} ACM Symp. on Operating System Principles (SOSP)*, 2011.

[134] Hin-Sing Siu, Yeh-Hao Chin, and Wei-Peng Yang. Byzantine agreement in the presence of mixed faults on processors and links. *Parallel and Distributed Systems, IEEE Transactions on*, 9(4):335–345, Apr 1998.

[135] K.P. Smith, B.T. Blaustein, S. Jajodia, and L. Notargiacomo. Correctness criteria for multilevel secure transactions. *Knowledge and Data Engineering, IEEE Transactions on*, 8(1):32–45, Feb 1996.

[136] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: A fast and scalable cryptocurrency protocol. Cryptology ePrint Archive, Report 2016/1159, 2016.

[137] Yonatan Sompolinsky and Aviv Zohar. Phantom: A scalable blockdag protocol. Cryptology ePrint Archive, Report 2018/104, 2018.

[138] Yee Jiun Song and Robbert Renesse. Bosco: One-step Byzantine asynchronous consensus. In *22^{nd} Int'l Symp. on Distributed Computing*, DISC '08, pages 438–450, Berlin, Heidelberg, 2008. Springer-Verlag.

[139] João Sousa, Alysson Bessani, and Marko Vukolic. A Byzantine fault-tolerant ordering service for the Hyperledger Fabric blockchain platform. In *48th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN)*, pages 51–58, June 2018.

[140] Matthew Spoke and Nuco Engineering Team. Aion: Enabling the decentralized internet. `https://aion.network`, July 2017.

[141] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, pages 149–160, August 2001.

[142] David Sutherland. A model of information. In *9th National Security Conference*, pages 175–183, Gaithersburg, Md., 1986.

[143] Doug Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89, December 2013.

[144] L. Torvalds and J. Hamano. Git distributed version control system. `https://git-scm.com`, 2010.

[145] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, February 2015.

[146] Werner Vogels. Eventually consistent. *Comm. of the ACM*, 52(1):40–44, January 2009.

[147] Cheng Wang. Alephium: a scalable cryptocurrency system based on blockflow. `https://alephium.org`.

[148] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[149] Joel Weinberger, Francois Marier, Devdatta Akhawe, and Frederik Braun. Subresource integrity. W3C recommendation, W3C, June 2016. `http://www.w3.org/TR/2016/REC-SRI-20160623/`.

[150] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. `https://polkadot.network`.

[151] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 279–294, New York, NY, USA, 2015. ACM.

[152] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM*

SIGSAC Conference on Computer and Communications Security, CCS '18, pages 931–948, New York, NY, USA, 2018. ACM.

[153] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop (CSFW)*, pages 29–43, June 2003.

[154] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. on Computer Systems*, 20(3):283–328, August 2002.

[155] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *5th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 293–308, 2008.

[156] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 263–278, New York, NY, USA, 2015. ACM.

[157] Lantian Zheng. *Making distributed computation secure by construction*. PhD thesis, Cornell University, Ithaca, New York, USA, January 2007.

[158] Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *18th IEEE Computer Security Foundations Workshop (CSFW)*, pages 272–286, June 2005.

[159] Lantian Zheng and Andrew C. Myers. A language-based approach to secure quorum replication. In *9th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, August 2014.